IOS函数响应型编程

Ash Furrow

目錄

iOS的函数响应型编程

Functional reactive programming introduction using ReactiveCocoa - By AshFurrow

本书翻译自FunctionalReactiveProgrammingOniOS

Gitbook地址

> 知识是人类进步的阶梯

翻译，喵～

译者为: kevinHM

如果在阅读过程中发现有什么问题，请到这里(本书在Github上的地址)开issue,我会尽快改正。

以下为本书的目录索引：

致谢

感谢 Justin Spahr-Summers、Josh Abernathy、Dave Lee 以及整个ReactiveCocoa团队开发出这样一个伟大的框架并维护其开发者社区。他们的工作使得我以及其他无数的开发者们受益匪浅！

感谢在本书的翻译过程中，老婆和家人的支持与理解.

理念

这将是高屋建瓴的一章。“为什么?!”你想，"尼玛！我以为这是一本关于编程的书,你特么跟我讲理念，我要拿回我的钱!"。稍安勿躁。。。这本书的受众是那些想要在编程中找到更好方式的开发者。那么我们现在首先要谈谈为什么我们想要更好的方式。

对向来以‘懒惰’著称的程序员们来说，为什么我们要选择改进？唯一可以理解的是，提高我们的技能让我们可以更‘懒’。。。我们希望用更少的代码来完成更多的任务。函数式反应型编程可以帮助我们达成这些目标，但它同时也意味着我们必须跳出自己的舒适区去接受函数式编程的洗礼。

所有的程序都是为了完成某些任务。大多数程序员所受的训练都是命令式编程。这种模式依赖于他们希望自己的程序如何来完成这些任务：开发者编写很多的指令来修正程序的状态；如果开发者在正确的位置上编写了正确的指令，那么程序将会正确地完成任务。

这听起来好平凡。。。

为什么编程时我们思考问题的方式都停留在“怎么做”这个点上? 因为计算机实际上是以一条条命令来工作的，CPU的程序计算器尽职尽责，按部就班：读取(怎么做的指令)---> 执行--->读取--->执行。。。所以理所当然的，我们只要告诉他们“怎么做”就好了(即命令式编程)。。。多么无聊啊。

与此相反，声明式编程(DeclarativeProgramming)将程序员们从纷繁复杂的对如何完成某些任务的细枝末节的流程中解放出来，将关注点集中在任务到底“是什么”而非实现任务的流程。声明式编程(DeclarativeProgramming)是命令式编程之外的几种编程范式的一个总称，我们将在稍后讨论。

维基百科:

> 声明式编程（英语：Declarative programming）是一种编程范型，与命令式编程相对立。它描述目标的性质，让电脑明白目标，而非流程。声明式编程不用告诉电脑问题领域，从而避免随之而来的副作用。而指令式编程则需要用算法来明确的指出每一步该怎么做。

函数式反应型编程是声明式编程的子编程范式之一，这是本书要讨论的主要内容。

函数式反应型编程

函数式反应型编程是两个声明式编程的子范例(函数式+反应式)的组合。这里我们先来理解反应式编程，因为它非常简单。

反应式编程在表处理方面十分强悍。假设我们有一个表格A:她是用来纪录其他两个表格(表格B、表格C)的和。当表格B或C当中任意一个值发现变化时，这些变化都会通过表实时改变表格A的值。总之，我们定义好了A是B和C的和，不管发生了什么,A会一直响应B或C的变化,永远都是B与C的和。

接下来我们来定义函数式编程。说实话很难准确定义它。任何试图通过Google这个词来了解它的人都会得到这样一个答案：函数范式是一个框架，可以用来构建我们的程序。函数式编程的核心是：在你的开发语言中函数本身是一个对象，且是所有类对象中的一等公民。

函数式编程中，对于同样的输入，一个函数f始终会给出同样的输出，不存在'可变的状态'。这听起来有点不可思议，我们可都是依靠状态的多变性来编写程序啊。在这个给变量赋值之后就不可以重新赋值的世界里，想想都觉得不可思议。函数式编程在很多方面显得不太实用。很多编程涉及到用户的输入、网络输入/输出等等，都不太容易使用函数范式来构建。这也是为什么函数式编程作为函数式反应型编程的一部分而出现的原因。因为函数式反应型编程是命令行编程与函数式编程两者相互妥协的最佳平衡点。她让我们有鱼与熊掌兼得的意思。

函数式反应型编程在处理用户输入时，就像是随着时间的改变而改变其结果的函数。有鉴于此，前面我们谈到的函数f，被假定为输入相同的参数就会返回一样的值，但如果参数是时间，则f就不会返回相同的值，因为时间一直在变化。这是一种‘欺骗’行为，但请记住，我们正在构建一个框架，在这个框架里面，我们都被允许实施这种'欺骗'行为,这就是函数式反应型编程。

结论

本章讨论的几个关键点：

- 学习函数式反应型编程，我们将会更加高效。
- 声明式编程把我们从关注业务的实现细节中解脱出来，用更多的时间关注业务本身。
- 函数式反应型编程是函数式与反应式编程的结晶。

我是一个实用主义者。我们所有的开发者们都是在实践中完成自己的作品的。因此我想尽可能少的占用你的时间来讲述理念的东西，在下一章节，我们将深入探讨代码实现。

用RXCollections进行函数式编程

这是一本关于函数响应式编程的书，对吗?!

好吧,就像我们在学会跑步之前必须先学会走路一样，在高效地进行函数响应式编程之前，我们得学会怎么样进来函数式编程。

高阶函数

函数式编程的一个关键的概念是"高阶函数"。从维基百科的解释来看，一个高阶函数需要满足下面两个条件:

- 一个或者多个函数作为输入。
- 有且仅有一个函数输出。

在Objective-c中我们经常使用block作为函数。我们不需要跋山涉水地去寻找'高阶函数'，实际上，Apple为我们提供的Foundation库中就有。考虑象下面这么简单的一个NSNumber 的数组：

```
NSArray * array = @[ @(1), @(2), @(3) ];
```

我们想要枚举这个数组的内容，利用数组元素来做些事情。

“好吧”，你说，“我将写一个for循环～”

住手吧，伙计，停止写for循环,好好看看我之前说的，我们可以用一个NSArray的高阶函数来代替。代码如下：

```
for (NSNumber *number in array) NSLog(@"%@",number);
```

。。。这个等同于下面的高阶函数:

```
[array enumerateObjectsUsingBlock:^(NSNumber *number, NSUInteger idx, BOOL *stop)
{
    NSLog(@"%@",number);
}];
```

"为什么?","这代码不是更多了吗?".

好吧，确实是这样，但这是通往函数式编程道路上的第一步:函数的启蒙教育。就像上一章节所说的，如何在只关注任务本身的前提下去完成任务？这只是为即将到来的便利付出的一点点代价，相信我。

实际上，高阶函数是很抽象的东西，我们所做的事情(命令式编程)基本上都可以用它来抽象。但Foundation中高阶函数的程度很低，要了解更多，我们不得不借助开源社区。

使用RXCollections

我的朋友RobRix使用OC写了一个优秀的高阶函数的库叫做RXCollections (译者注：目前这个项目作者已经停止维护，取而代之是RobRix的另外一个项目Reducers)

首先，我们需要一个可以展示的Xcode工程，创建一个新工程“Playground”。选择"Single View Application"作为模板。我们将在AppDelegate中展示绝大部分代码。在本书中，我将使用"FRP"作为类的前缀。

其次，我们需要在工程中导入RXCollections.我将使用Cocoapods导入这个库，这会让事情变得简单。使用如下命令以确保你的电脑里安装了最新的cocoapods。

```
sudo gem install cocoapods
```

终端出现提示的时候输入你的root密码。一旦cocoapods已经安装好了，使用 `cd` 导航到刚刚新建的工程目录下，并在终端输入如下指令:

```
pod init
```

这将会在当前目录下为你生成一个新的文件 `Podfile` .内容大致如下:

```
#Uncomment this line to define a global platform for your project
#platform :ios, "6.0" (这里为m.n 取决于工程的设置)

target "Playground" do

end

target "PlaygroundTests" do

end
```

用你最习惯的文本编译器(我猜是Vim),取消 `#platform :ios,"6.0"` 的注释标示，并添加 `pod 'RXCollections' , '~> 1.0'` 到"Playground"下。

```
platform :ios, "6.0"

target "Playground" do

pod 'RXCollections', '~> 1.0'

end

target "PlaygroundTests" do

end
```

好了！保存并退出编辑器，回到终端并输入:

```
pod install
```

这将导入RXCollections到工程中，同时为你提供一个新的xcode workspace文件。关闭当前xcode工程，用刚刚生成的workspace文件打开工程。

在Appdelegate.m文件中引入如下头文件:

```
#import <RXCollections/RXCollection.h>
```

在`application:didFinishLaunchingWithOptions:`方法中，创建一个我们之前讲到的数组。

```
NSArray * array = @[ @1, @2 , @3 ];
```

好了，万事具备，开始染手函数式编程！

高阶映射

我们要学习的第一个高阶函数是'映射[map]'.映射是在函数的层次上把一个列表变成相同长度的另一个列表，原始列表中的每一个值，在新的列表中都有一个对应的值。如下所示是一个平方数的映射：

```
map(1,2,3) => (1,4,9)
```

当然，这只是一个伪代码，一个高阶函数会返回另外一个函数而不是一个列表。那么我们要如何利用RXCollections呢?

我们这么来用rx_mapWithBlock:方法：

```
NSArray * mappedArray = [array rx_mapWithBlock:^id(id each){
    return @(pow([each integerValue],2));
}];
```

这将会达成上面伪代码所完成的任务，如果我们打印出`array`的日志，我们将会看到如下内容:

```
(
    1,
    4,
    9
)
```

简直完美!请注意`rx_mapWithBlock:`并不是一个真正的函数映射，因为他不是技术上的高阶函数(她没有返回一个函数)。后面提到的库(RAC)已经解决了这一点,在下一章我们将看到映射是如何在ReactiveCocoa的上下文中工作的。

注意`rx_mapWithBlock:`在没有对原数组元素进行任何修改的前提下返回了一个新的数组，这里Foundation的类真的是非常好用的一个例子，因为他们的类默认就是不可变的。

想象一下，往常(命令式编程)为了完成这个任务，我们不得不写下这样的代码:

```
NSMutableArray *mutableArray = [NSMutableArray arryaWithCapacity:array.count];
for (NSNumber *number in array) [mutableArray addObject:@(pow([number integerValue], 2))]

NSArray *mappedArray = [NSArray arrayWithArray: mutableArray];
```

代码显然更多，而且还有一个无用的局部变量`mutableArray`污染了我们的作用域，简直是个毛线！

所以当你想把一个列表里的元素转化为另一个列表的元素时，你就能体会到映射的强大。

高阶过滤

谈到ReactiveCocoa，我们要使用的另一种关键的高阶函数就是过滤器。一个列表通过过滤能够返回一个只包含了原列表中符合条件的元素的新列表，具体我们来看实践中的例子:

```
NSArray *filteredArray = [array rx_filterWithBlock:^BOOL(id each){
    return ([each integerValue] % 2 == 0);
}]
```

过滤后，现在`filteredArray`等于`@[ @2 ]`.如果没有这样的抽象方法(即高阶过滤)，我们不得不像下面这样来完成工作:

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithCapacity: array.count];
for ( NSNumber * number in array ){
    if ( [number integerValue] % 2 == 0 ){
        [mutableArray addObject:number];
    }
}
NSArray *filteredArray = [NSArray arrayWithArray:mutableArray];
```

有点明白了,对不对? 你可能像上面这样子写代码写了成百上千次。我们每一天的工作中涉及到类似这种高阶映射或者高阶过滤的事情有多少? 非常多！通过使用像高阶过滤、高阶映射类似的高阶函数，我们能够把这种繁琐又乏味的任务抽象出来，轻松工作，轻松生活。。。

高阶折叠

Flod 是一个有趣的高阶函数ー她把列表中的所有元素变成一个值。一个简单的高阶折叠能够用来给数值数组求和。

```
NSNumber * sum = [array rx_foldWithBlock:^ id (id memo , id each){
    return @([memo integerValue] + [each integerValue]);
}];
```

输出的值为@6.数组中的每一个元素按顺序执行上述合并规则: `[memo integerValue] + [each integerValue]`,其中memo参数纪录的是上一次合并后的结果，其初始值为零。这还不是很有趣，有趣的是我们还能给 `memo` (这个参数的泛称)赋初始值:

```
[[array rx_mapWithBlock:^id (id each){
        return [each stringValue];
    }] rx_foldInitialValue:@"" block:^id (id memo , id each){
        return [memo stringByAppendingString:each];
}];
```

代码的结果:@“123”. 我们来分析一下这是怎么做到的. 首先我们对数组中的所有NSNumber对象做了映射，把他们变成了NSString对象，然后我们实现了一个高阶折叠，并给了 `memo` 变量一个空字符串。

在没有RXCollections的情况下能得到这样的结果吗？当然可以。但这是一个明确的"是什么，而不是如何"的解决问题的方法。这种方法可以让我们不必跟CPU一样去想"这一步要如何，下一步要如何"类似这样的事情。写代码的时候如此，读代码的时候更是如此(意:更多地关注任务是什么，要达成什么目标)

性能

这一章有关函数式编程的事例代码可能会让你开始担心性能的问题。例如，在一个长数组中，给每个元素创建一个过渡的字符描述并把他们追加到前面的结果中去，比起命令式编程来说，可能需要消耗更长的时间。

这可能是个问题，但幸运的是，现在的计算机(甚至iPhone手机)性能已经足够强大，在大多数情况下，这种性能损耗是无关紧要的，况且当这种损耗变成一个性能瓶颈的时候，你随时都可以回头去优化她让她更加高效。CPU的时间很廉价，但是你的时间是很宝贵，因此牺牲CPU的时间会是更好的选择。

总结

在过去的章节中，我们使用RXCollections后不需要额外的可变变量就可以在列表上进行操作，虽然RXCollections可能隐式地生成了这样的可变变量来完成任务，但是这不是我们要关心的，因为它已经为我们抽象出了这样的方式，通过:mapping\filtering和folding这种方式让我们不必在意实现任务的步骤。(当然，这并不是说，我们不应该熟悉RXCollections的源码，只是告诉你不必按部就班地去完成任务了)

在最后的章节中，我们也看到了，使用链式操作一次可以输出一个更为复杂的逻辑操作的结果。下一章我们将谈论更多的有关链式操作的内容———实际上，它是ReactiveCocoa中的重要语法之一。

下一章，我们将要讨论更多的有关映射、过滤及折叠相关的内容。我们不仅仅局限于将高阶函数运用在列表的操作上，我们也将用她们来操作流(译者注:一切皆文件，文件以流的形式传播，也就是说一切的操作都可以使用高阶函数)，还会介绍其他的高阶函数。

ReactiveCocoa 简介

上一章我们学习了函数方法:map、filter以及fold,我们将再次熟悉她们。但是这一章是围绕着ReactiveCocoa和函数响应式编程来展开的，学习之前需要做一点点补充说明。

引入ReactiveCocoa

ReactiveCocoa有两种引入的方式:使用CocoaPods或者作为项目的一个字模块(直接拽入项目中)。ReactiveCocoa官方是不支持CococaPods的，但是开源社区提供了这样的服务，我们可以使用她。如果你乐于让ReactiveCocoa作为一个子模块引入到项目中，你可以下载2.x版本并根据官方的介绍来配置她。

使用CocoaPods来引入ReactiveCocoa：打开前面我们创建的 `Podfile` 文件，并删除 `RXCollections` 行，用 `pod 'ReactiveCocoa', '2.0'` 替代掉。你的 `Podfile` 文件看起来应该是这样的:

```
platform :ios, "6.0"
target "Playground" do
pod 'ReactiveCocoa' , '2.0'
end

target "PlaygroundTests" do

pod 'ReactiveCocoa' , '2.0'
end
```

注意：我们使用的是'2.0'版的ReactiveCocoa而非最新的。重新运行 `pod install` ,将从项目中移除 `RXCollections` 并引入 `ReactiveCocoa` 。项目中任何 `#import <RXCollections/XXXX>` 的地方都会编译报错，请把他们也移除。

这一章里面，我们将把代码写在 `ViewController` 的实现文件中，而不是在 `AppDelegate` 中，所以现在请打开ViewController的实现文件。不要忘记把ReactiveCocoa引入进来 `#import <ReactiveCocoa/ReactiveCocoa.h>` 。

流和序列

流是值的序列化的抽象，你可以认为一个流就像一条水管，而值就是流淌在水管中的水，值从管道的一端流入从另一端流出。当值从管道的另一端流出的时候，我们可以读取过去所有的值，甚至是刚刚进入管道的值(即当前值)。接下来让我们拭目以待！呃，值的序列化，那是什么鬼？以我们当前的认知水平来说，她就像是一个数组，一个列表。事实上，使用 `rac_sequeuece` 我们能够轻松地将数组转化为一个流:

```
NSArray *array = @[ @1, @2, @3 ];
RACSequence * stream = [array rac_sequence];
```

等一下！ `Sequences` ？我以为我们在处理 `Stream` ？好吧，说明一下， `Sequences` 是两种特定类型的流的一种，实际上， `RACSequence` 是一个 `RACStream` 的子类。 我们能用流做什么呢?好吧，我将使用流来展示上一章中提到的例子。应用在平方数映射上:

```
[stream map:^id (id value){
    return @(pow([value integerValue], 2));
}];
```

注意，跟数组一样，流不能包含nil元素。[译者注:NSArray中以nil作为结束标示，stream也一样]。 非常好！但是流映射后还是流，我们怎么样才能得到数组呢？幸运的是, `RACSequence` 有一个方法返回数组: `array` 。

```
NSLog(@"%@",[stream array]);
```

这会打印映射后的数组。比起直接使用 `RXCollections` 这多出了几个步骤，但这里我只想说明使用流也可以达成任务。

当然，我们可以合并上面的方法调用来避免污染变量的作用域.

```
NSLog(@"%@",[[[array rac_sequence] map:^id (id value){
                    return @(pow([value integerValue], 2));
                }] array]);
```

总的来说,我们做了这样的事情:

- 将数组转化成一个序列类型的流。
- 对流进行映射得到一个新的流。
- 将新的流转为数组。

序列，默认情况下是延迟加载的(也称：懒加载或被动加载)，是 `pull-driven` 的，在他们被生成的时候就会提供确切的值，而数组方法会强制给序列的每一个成员赋值。

我们来看一下 `filtering` 。为了使用ReactiveCocoa来过滤我们的数组，我们需要再一次把它序列化以便

于使用过滤。

```
NSLog(@"%@", [[[array rac_sequence] filter:^BOOL (id value){
                        return [value integerValue] % 2 == 0;
                    }] array]);
```

最后看一下怎么让一个序列流合并为单个值(`folding`)：

```
NSLog(@"%@",[[[array rac_sequence] map:^id (id value){
                    return [value stringValue];
                }] foldLeftWithStart:@"" reduce:^id (id accumulator, id value){
                    return [accumulator stringByAppendingString:value];
            }]);
```

这种情况下，我们在序列上进行了链式调用，当我们讨论下一节'信号'的时候，(链式调用)是一个关键的概念。

ReactiveCocoa具有左折叠和右折叠的概念。左折叠时折叠算法将从头到尾遍历数组，反之称为右折叠。这样的命名(即左、右折叠)暗示了编程语言对列表的理解，这种概念在Objective-C中是没有的。

确定你现在已经理解了到此为止我们所说的内容，这对后面将要进行的讲解非常重要。

信号

信号是另一种类型的流。与序列流相反，信号是 `push-driven` 的。新的值能够通过管道发布但不能像 `pull-driven` 一样在管道中获取，他们所抽象出来的数据会在未来的某个时间传送过来。

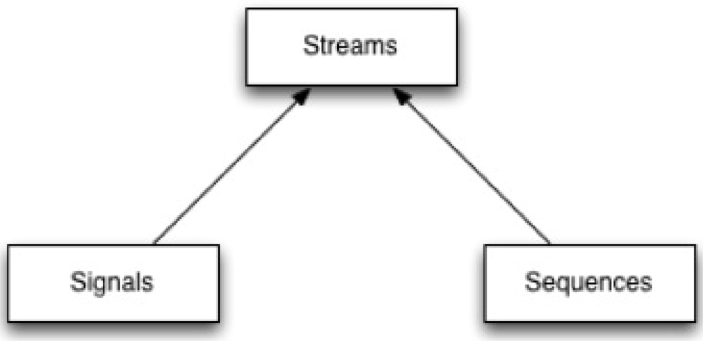
这里需要理解两个概念: `pull-driven` 和 `push-driven` .

> Push-driven means that values for the signal are not defined at the moment of signal creation and may become available at a later time (for example, as a result from network request, or any user input).
>
> Push-driven : 在创建信号的时候，信号不会被立即赋值，之后才会被赋值(举个栗子：网络请求回来的结果或者是任意的用户输入的结果)
>
> Pull-driven means that values in the sequence are defined at the moment of signal creation and we can query values from the stream one-by-one.
>
> Pull-driven : 在创建信号的同时序列中的值就会被确定下来，我们可以从流中一个个地查询值。

信号发送三种类型的值： `Next Values` 代表了下一个发送到管道内的值。 `Error Value` 代表 `signal` 无法成功完成,一般很少见，我们会在下一章学习怎么使用她们。 `Completion Values` 代表 `signal` 成功完成，我们也会在下一章来学习。这里要注意的是：

> 一个事情响应中，一个 `signal` (信号)发送了一个 `Error value` 或者一个 `Completion Value` 后，就不会再发送任何其他的 `value` . 错误或者成功将只会发送其中一个，绝不会有两个同时发送的情况！

信号是ReactiveCocoa的核心组件之一。ReactiveCocoa为UIKit的每一个控件内置了一套信号选择器。例如，UITextField就有一个 `rac_textSignal` ,UITextField中每一次按键的响应都会通过它发送出去。下一章我们会学习如何使用信号来执行任务。



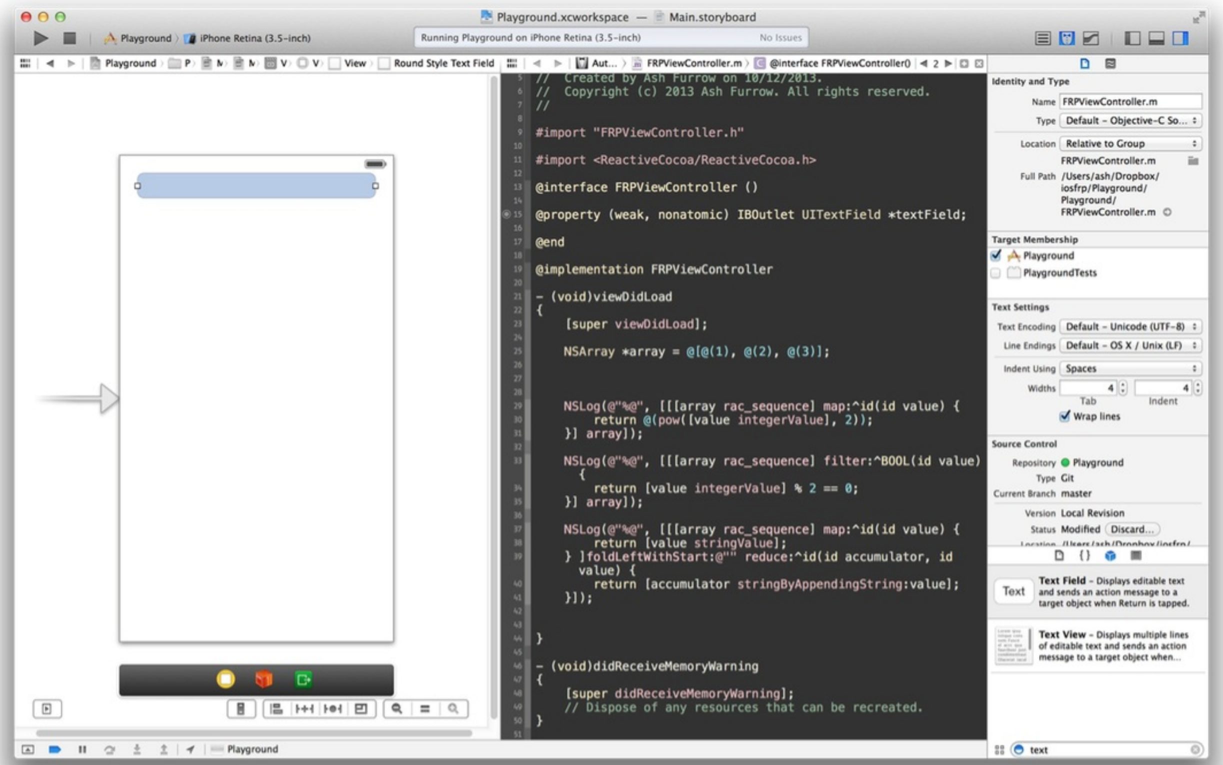
Class diagram

信号也可以被链接(链式调用)和转化。通过映射或者过滤一个流得到的新的流也可以随后被映射、被过滤，进行所有你能想到的各种操作。下一章我们将了解更多这方面的内容。

订阅

当你随时都想知道某一个值的改变时(不管是next、error或者completion),你就会订阅流---一种最常见的signal.使用信号通常都会有副作用，比如下面这个例子。

我们添加一个textfield控件到viewController's View上，这里我使用storyboard来做，你怎么做随你喜好。

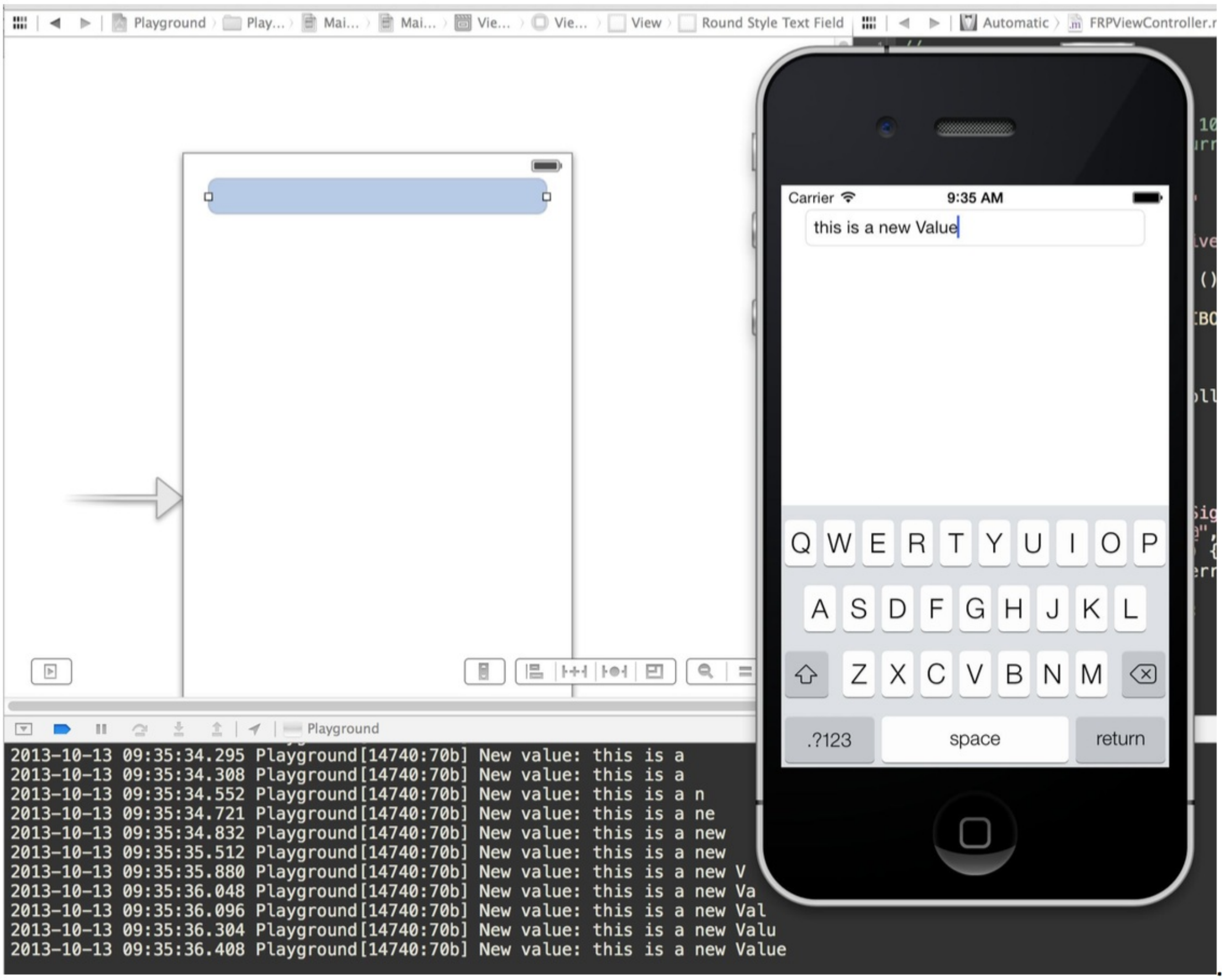


Adding a text field

在ViewDidLoad中添加如下代码，订阅textfield的rac_textSignal。

```
[self.textField.rac_textSignal subscribeNext:^(id x){
    NSLog(@"New Value: %@",x);
} error:^(NSError * error){
    NSLog(@"Error : %@", error);
} completed:^{
    NSLog(@"Completed.");
}];
```

创建并运行应用程序，在textField上输入一些内容。每一次每一个新的值输入到textField中，这个 `Next value` 就会下发到管道中，然后我们的订阅块就会被执行。



有趣的是，这个特殊的信号不会发送错误值，仅仅在释放的时候发送一个完成值，所以这两个订阅块通常不会被调用。我们可以使用RACSignal上的一个简便的方法 `subsribeNext:` 来简化我们的代码:

```
[self.textField.rac_textSignal subscribeNext:^(id x){
    NSLog(@"New Value: %@", x);
}];
```

看吧，少了很多代码!

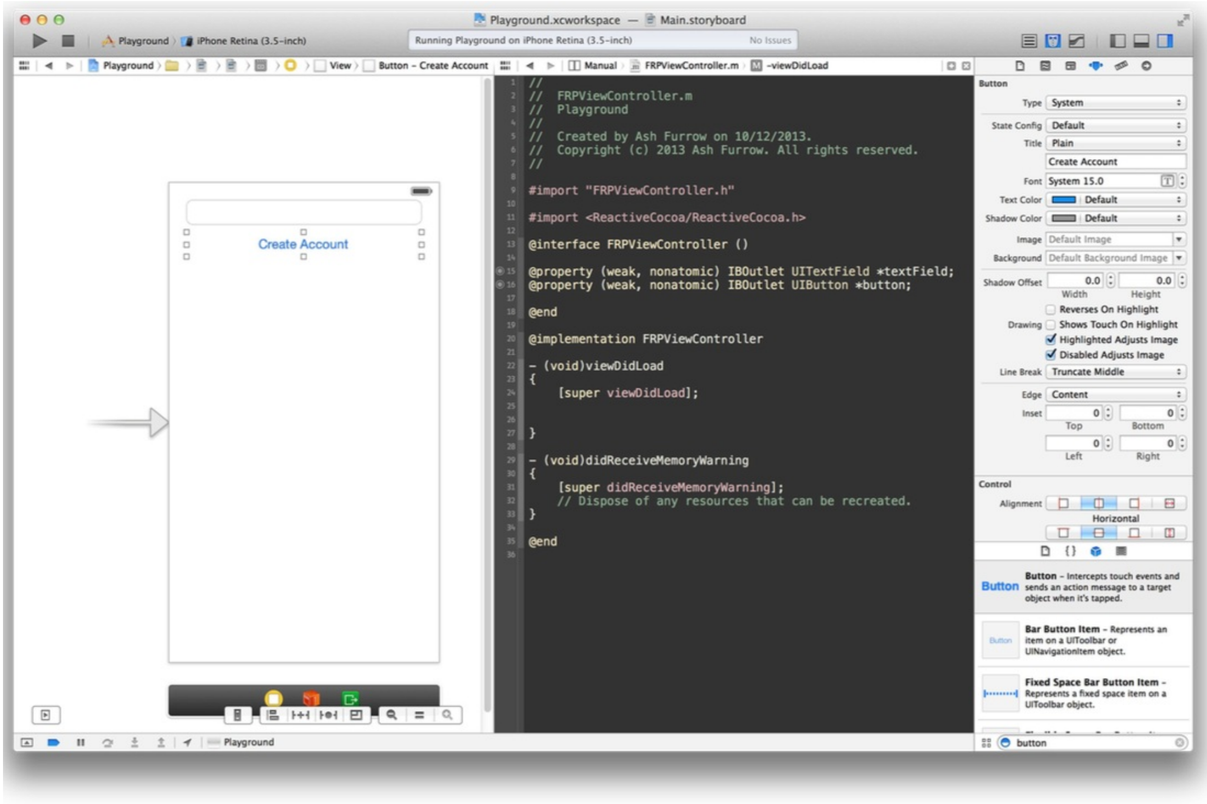
当你订阅一个信号时，实际上你创建了一个'订阅者'，她是自动保留的，并同时保留她订阅的信号，你也可以手动配置这个'订阅者'，但这不是一种典型的行为。下一章我们将会学习，当视图复用的时候(像CollectionViewCells 或TableViewCells)，如何去有效地配置信号。

状态推导

状态推导是ReactiveCocoa的另一个核心组件。这里并非指类的某个属性(设置一个新的值就代表状态发生了改变那样)，这里我们指的是把属性抽象为流。就拿前面的例子，我们为她增加状态推导。

假设我们的视图是用来创建账户的，我们只允许包含有'@'字符的Email地址，当且仅当，输入的用户名有效时使按键可用，同时我们也希望通过TextField中Text的颜色给用户提供反馈。

- 首先我们使用IBOutlet在视图上增加一个按键‘button’.



Added a button

- 其次我们将button的enable属性与我们创建的信号绑定。

```
RAC(self.button, enabled) = [self.textField.rac_textSignal map:^id (NSString *value){
    return @([value rangeOfString:@"@"].location != NSNotFound);
}];
```

请注意，稍候将看到我们如何使用buttons的命令来更好地约束她的enable属性。

`RAC()` 宏需要两个参数:‘对象’以及这个对象的某个属性的'keyPath'。然后将表达式右边的值和'keyPath'做一个单向的绑定，这个值必须是NSObject类型，所以我们会把boolean量封装成NSNumber。

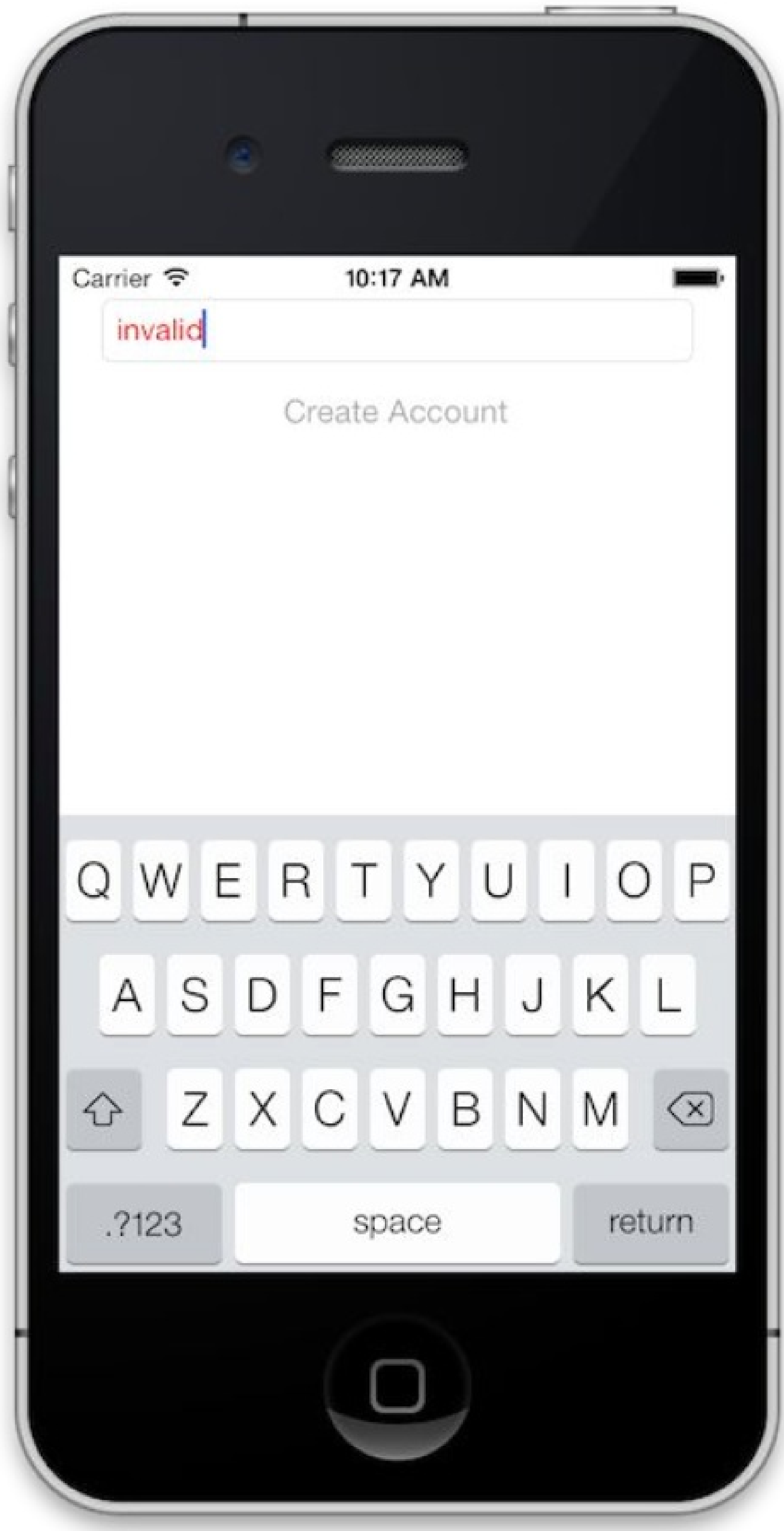
但是，文本的颜色怎么办?实际上我们在这个基础上做一点点重构就可以了。

```
RACSignal * validEmailSignal = [self.textField.rac_textSignal map:^id (NSString *value){
    return @([value rangeOfString:@"@"].location != NSNotFound);
```

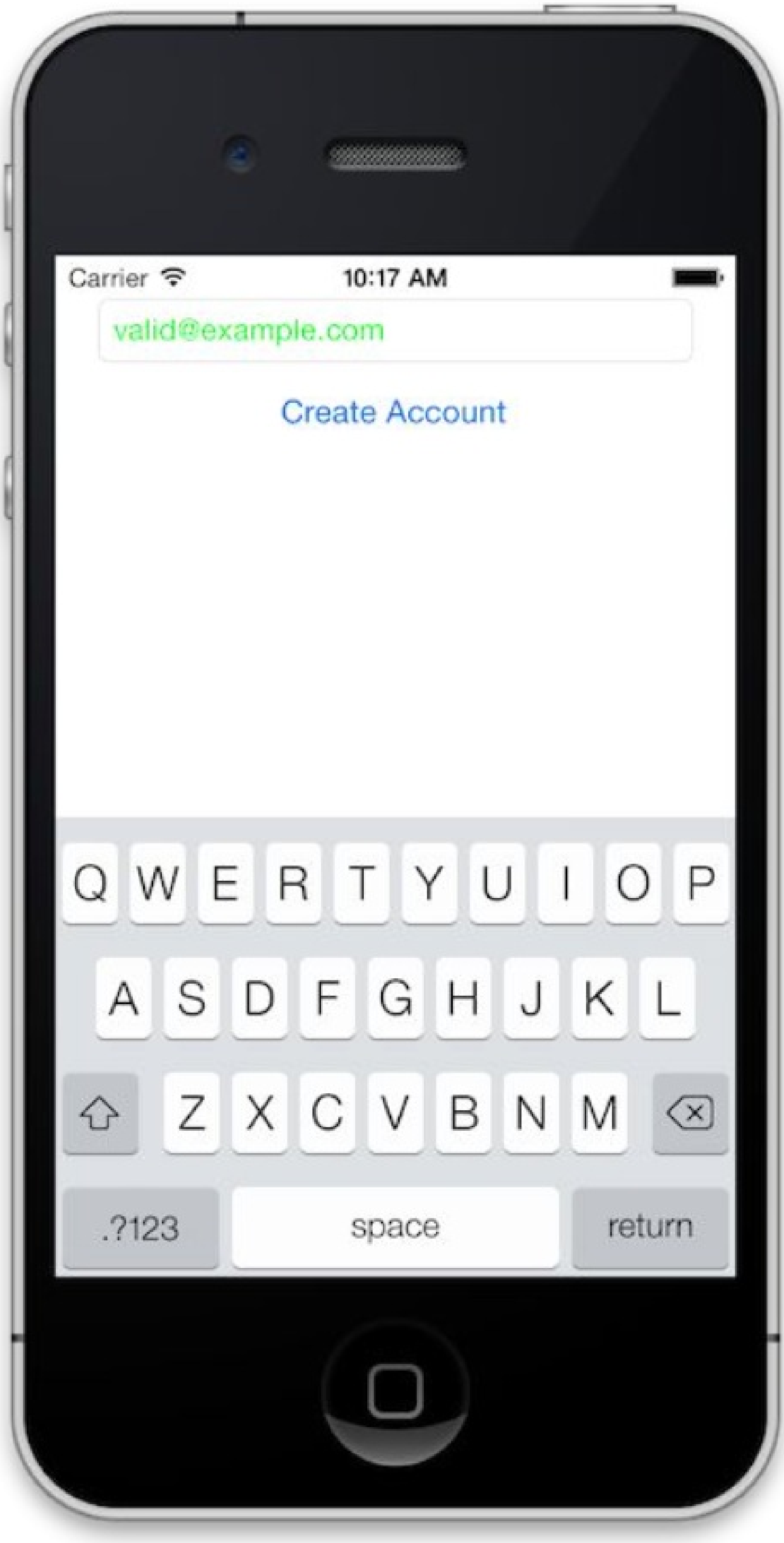
```
}];

RAC(self.button, enabled) = validEmailSignal;

RAC(self.textField, textColor) = [validEmailSignal map: ^id (id value){
    if([value boolValue]){
        return [UIColor greenColor];
    }else{
        return [UIColor redColor];
    }
}];
```



Invalid email address



Valid email address

很好！看到我们怎样复用validEmailSignal吗？这在ReactiveCocoa中是非常常见的用法。在viewDidLoad方法之外，我们也不用写任何代码，这也很常见。

指令

上一节，我们绑定UIButton的enabled属性并不是最佳实践，因为UIButton增加了一个ReactiveCocoa的类和一条指令。在这一节中我们将介绍ReactiveCocoa的指令。实际上button的rac_command可以为我们监控enabled属性。 应用一段ReactiveCocoa的文档:

> 指令，RACCommand类的代表，创建并订阅动作的信号响应，可以很容易地实现一些用户与应用交互时的边界效果。
>
> 指令(行为触发的)通常是UI驱动的，比如按键的点击。指令也可以通过信号自动禁用，这种禁用状态呈现在UI上就是禁用与该指令相关联的任何操作。

当你想要一次用户交互发送一个信号来响应的时候指令就很有用。指令信号对订阅了指令的这个信号而言，她之后的输出都被指令信号所处理。这有一点点混乱，在第五章我们会看到一些指令相关的实践。

现在我们用下面的代码来替代之前的在button上绑定enabled属性的代码

```
self.button.rac_command = [[RACCommand alloc] initWithEnabled:validEmailSignal
                                                signalBlock:^RACSignal *(id input){
                                                    NSLog(@"Button was pressed.");
                                                    return [RACSignal empty];
                                                }];
```

任何时候button被点击就会执行signalBlock，rac_command属性会监控使能信号validEmailSignal和button的enabled属性。(实际上，如果我们保留原来的代码，新加这一段会引起重复绑定一个属性的错误)。

另外，这里返回的[RACSignal empty]是什么东西? 呃。。。这里我们需要返回一个信号让属于RACCommand的 `executionSignal` 管道(pipe)下发出去。这个信号代表button按下时一些任务需要被处理。在这个处理信号没有返回一个'complete value'('empty '会立即返回一个'complete value')之前button将会保持不可用状态.因为这个例子中我们只是打印了一下，所以这里我们只返回一个empty信号。在第五章我们将继续讨论RACCommand及其用途。

RACSubject

RACSubject是一个有趣的信号类型。在'ReactiveCocoa'的世界中，她是一个可变的状态。她是一个你可以主动发送新值的信号。出于这个原因，除非情况特殊，我们不推荐使用她。

下一章我们将学习RACSubject们如何嫁接non-reactivecocoa和reactivecocoa的代码。

热信号与冷信号

信号是典型的懒鬼，除非有人订阅他们，他们是不会启动并发送的。每增加一个订阅，它们都会重复地多发送一个信号。鉴于用户操作的琐碎性，这种设计是可接受的。实际上，在ReactiveCocoa的命名法则中，这种信号被称为'冷(信号)'。

有的时候我们希望让信号立即工作(不需要中间这么繁琐的设置),ReactiveCocoa中称为'热(信号)'。这种信号用的非常少。

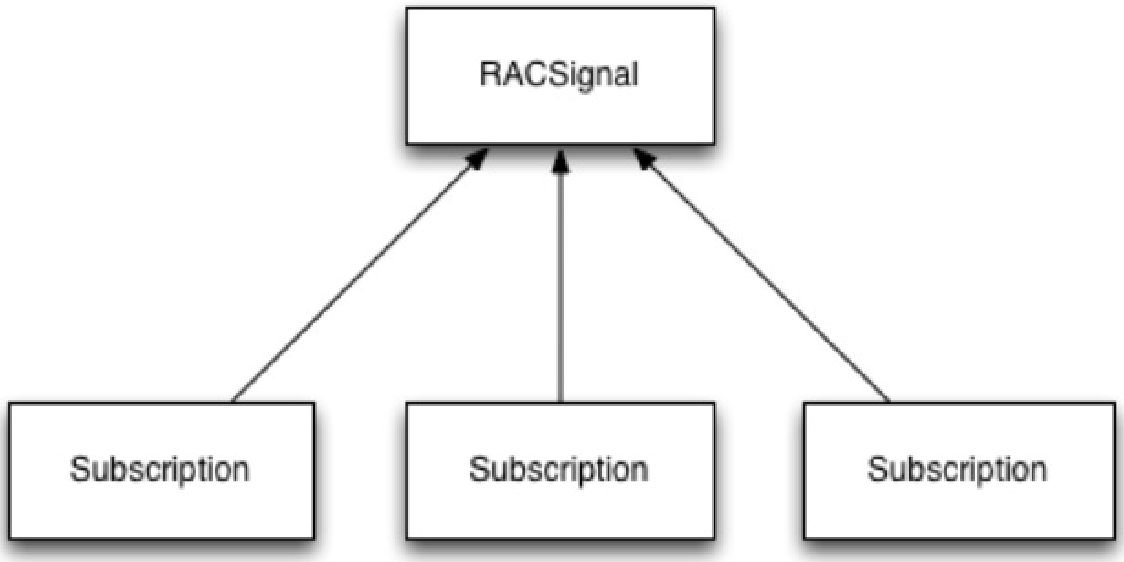
这两者的不同是很微妙的，在下一章我们将学习如何利用热信号。

组播

组播是用语多个订阅者共享一个订阅信号的术语。就像我们上一节所描述的那样，默认的情况下，信号是冷的。有时候，我们不希望一个冷信号在每一次被订阅时工作。这通常在边界效应、订阅所要执行的任务代价昂贵或者只能以其他方式在适当的时间执行时有这种需求。这时网络请求浮现在脑海中。。。

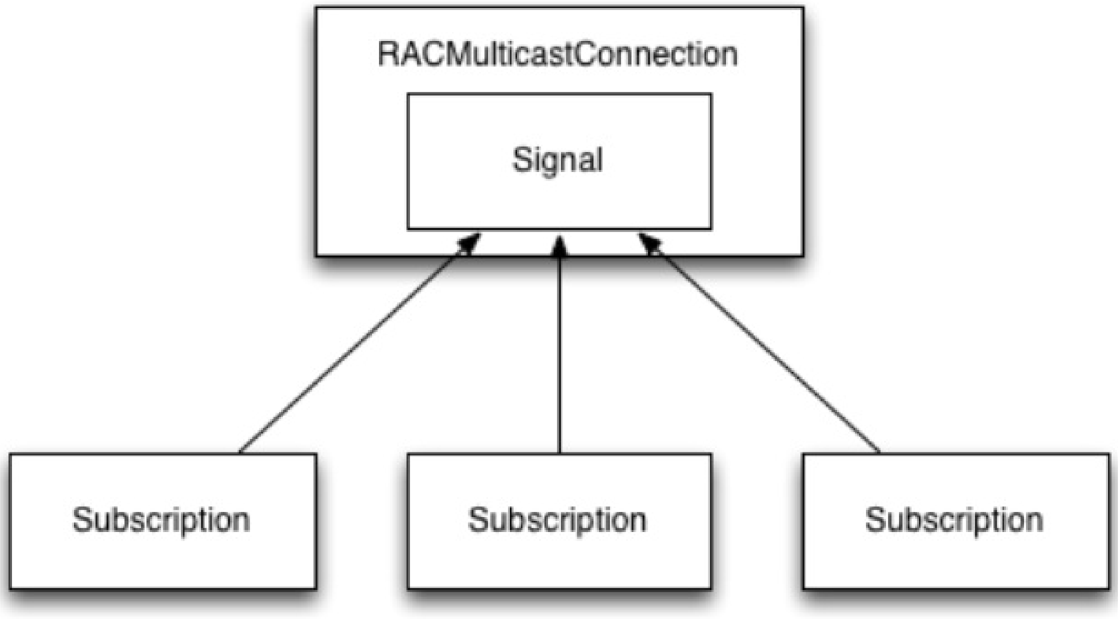
所以与其从这样的信号中创建一个 `RACMulticastConnection` ,不如使用 `RACSignal` 的 `publish` 方法或者 `multicast:` 方法。前者为您创建一个组播连接，后者也一样为您创建一个组播连接但需要一个 `RACSubject` 参数。当她被调用时这个RACSubject可以通过底层信号发送一个值出来。任何对这个值有兴趣的，都可以用这个从底层信号发送一个值到连接的信号来替代你提供的 `RACSubject` ，这个信号恰好就等同于你的这个 `RACSubject` .

为了说明这种不同，请参考下面的插图：



Multiple subscriptions

由于信号是冷启动的，每增加一个订阅者，她就会被执行一次。这种情况是我们不希望看到的，可以使用组播连接来改善。



Multicast connection

信号的组播连接订阅，当她传送一个新值的时候，是通过公共频道来传送给信号的。只要你喜欢你可以随意订阅这个信号，但这个信号在订阅相关的操作上有且仅会执行一次，不再像以前那样增加一个订阅者这个信号上就执行一次订阅相关的操作。

总结

这一章我们讨论了很多概念，在没有任何实践的基础上来理解这些东西，尤其是一些高级的概念，是比较困难的。下一章，我们将运用我们所获取的这些知识，使她们深入人心。我们将要诠释的不仅仅是我们在这里看到的这些，同时我们也将获得一些ReactiveCocoa的最佳实践。

ReactiveCocoa的实践

这一章中我们将第一次使用ReactiveCocoa来编写一个实际的应用。我们将创建一个叫做'500px'的简单应用。'500px'类似于'Flickr',但只有你满意的照片才会被存放在那里。我使用'500px'的API的原因有两点:

- 照片看起来非常棒
- 当我还在那里工作的时候，我为他们的API接口写了iOS的SDK，我很熟悉她。

这一章我们分三个部分来讲解：

- 首先将完成我们的App(FunctionalReactivePixels)的基本实现。
- 其次我们将添加一些新的视图控制器，做更多的数据加载，来进一步证实第一步的实现。
- 最后我们将重新审视这个应用程序，以消除更多的状态获取使用更多函数响应型编程的机会。

这一章非常有趣，当然由我亲笔完成。我们应用程序'FunctionalReactivePixels'最后的结果开源在Github上,不幸的是，创作这个App的中间一些过程并不会展现在最后的结果里，但是如果你一章一章跟着我来的话，应该会很好。

FunctionalReactivePixels的基础知识

FunctionReactivePixels将会是一个简单的观看'500px'中最受欢迎的照片的应用。一旦我们完成这一节，应用的主界面将会像下面这样：



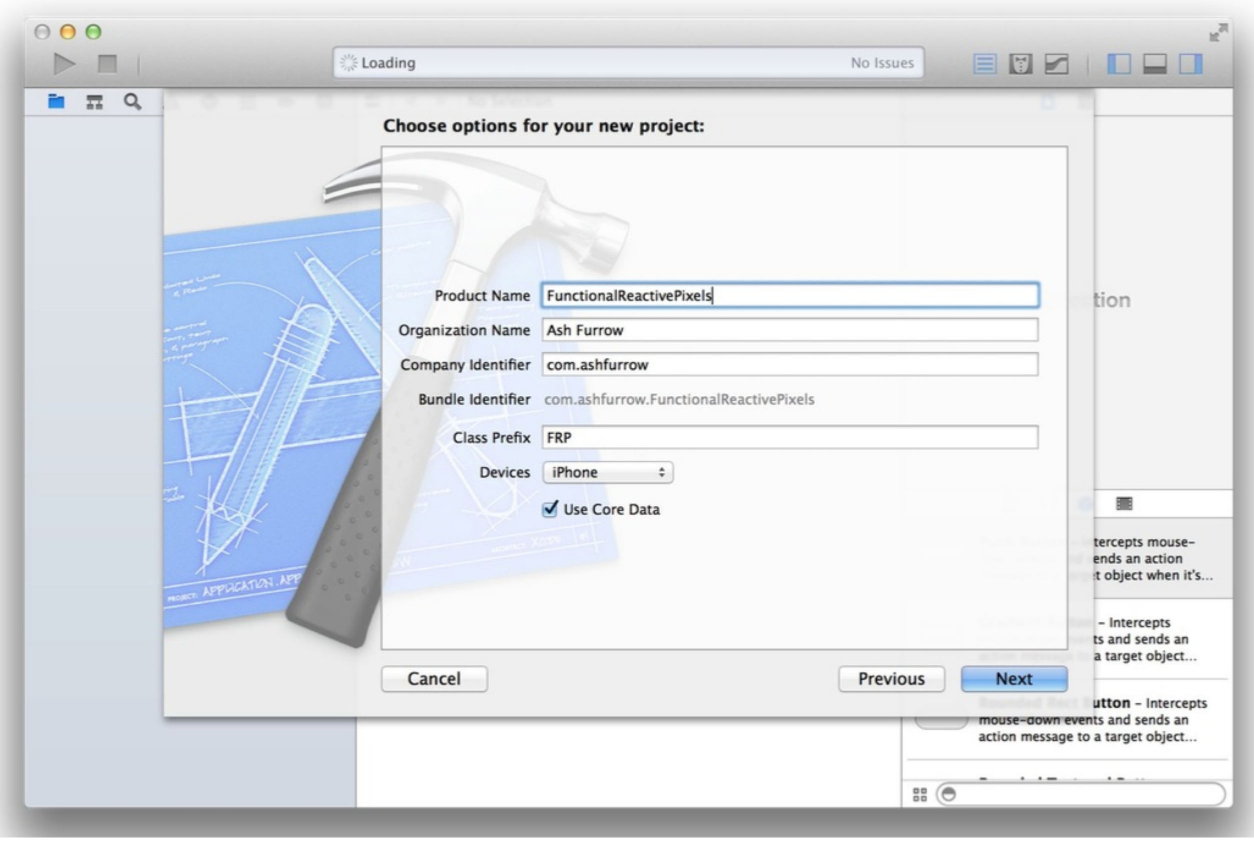
当然我们也可以像下图一样观看全屏模式下的图片。



这个App将使用Collection Views。如果你没有太多这方面的经验，也不需要太过担心---他们

(CollectionView)就像TableView一样，使用起来非常简单。如果你对UICollectionView感兴趣，可以阅读我的另一本书.

我们将使用CocoaPods来管理我们的依赖，现在创建一个新的工程。我喜欢使用空模版以便我可以完全控制viewController层级。



首先、我们将创建一个UICollectionViewController的子类FRPGalleryViewController.同时我们创建一个UICollectionViewFlowLayout的子类FRPGalleryFlowLayout.

```
#import the new flow layout's header in the view controller's implementation file and
#then override FRPGalleryViewController's init method

- (id)init{
    FRPGalleryFlowLayout *flowLayout = [[FRPGalleryFlowLayout alloc] init];
    self = [self initWithCollectionViewLayout:flowLayout];
    if(!self) return nil;
    return self;
}
```

这将初始化collection View的layout为我们自己的layout.这个flowlayout子类的实现非常简单，只需要设置一些属性就可以了。

```
@implementation FRPGalleryFlowLayout
- (instancetype)init{
    if (!(self = [super init])) return nil;

    self.itemSize = CGSizeMake(145,145);
    self.minimumInteritemSpacing = 10;
```

```
    self.minimumLineSpacing = 10;
    self.sectionInset = UIEdgeInsetsMake(10,10,10,10);

    return self;
}
@end
```

很棒!下一步，我们需要把Viewcontroller展现在屏幕上。为了实现这个，我们首先要在应用的application delegate的`application: didFinishLaunchingWithOptions:`方法。我们想要将collectionview Controller置于一个navigationController容器中：

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.rootViewController = [[UINavigationController alloc] initWithRootViewCont

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

很好!如果我们现在运行，我们将看到一个空视图。


Carrier
1:06 PM

我们来填充一些内容。创建一个Podfile文件,并填写如下内容:

```
platform :ios, "7.0"
target "FRP" do
    pod 'ReactiveCocoa', '~> 2.1.4'
    pod 'libextobjc', '~> 0.3'
    pod '500-iOS-api', '~> 1.0.4'
    pod 'SVProgressHUD', '~> 0.9'
end

target "FRPTests" do

end
```

下一章，我们将添加一些测试。现在运行`pod install`,然后打开Xcode通用的`workspace`文件。打开与编译头文件`FRP-Prefix.pch`(Xcode6之后，新建工程默认不加载pch文件，需要自己添加，Apple的最佳实践中已经不推荐使用全局的预编译pch文件)，然后添加下面的内容。这些语义会自动加载到项目的所有文件中。

```
//Pods
#import <ReactiveCocoa/ReactiveCocoa.h>
#import <500px-iOS-api/PXAPI.h>
#import <libextobjc/EXTScope.h>

//App Delegate
#import "FRPAppDelegate.h"
#define AppDelegate ((FRPAppDelegate *)[[UIApplication sharedApplication] delegate])
```

对于这样使用AppDelegate单例的用法，Saul Mora说：“每次看到你这么做，我家的狗都想死”。 但是这不是一本关于设计模式的书---这是一本关于ReactiveCocoa的书，所以我们可能要害死一些狗狗。。。

创建一个AppDelegate的属性来hold住500px API客户端

```
@property (nonatomic, readonly) PXAPIHelper * apiHelper;
```

在`application:didFinishLaunchingWithOptions:`方法中实例化这个变量。

```
self.apiHelper = [[PXAPIHelper alloc]
                    initWithHost:nil
                    consumerKey:@"DC2To2BS0ic1ChKDK15d44M42YHf9gbUJgdFoF0m"
                    consumerSecret:@"i8WL4chWoZ4kw9fh3jzHK7XzTer1y5tUNvsTFNnB"];
```

我提供了一对一次性消费的密钥---请不要疯到你也使用这对密钥，你可以申请自己的。

好了，我们差不多也该建立数据的加载了。我们需要一个数据模型来hold住我们的信息。我创建了下面的`FRPPhotoModel`。

```
@interface FRPPhotoModel : NSObject
```

```
@property (nonatomic, strong) NSString *photoName;
@property (nonatomic, Strong) NSNumber *identifier;
@property (nonatomic, strong) NSString *photographerName;
@property (nonatomic, strong) NSNumber *rating;
@property (nonatomic, strong) NSString *thumbnailURL;
@property (nonatomic, strong) NSData *thumbnailData;
@property (nonatomic, strong) NSString *fullsizedURL;
@property (nonatomic, strong) NSData * fullsizedData;


@end

@implementation FRPPhotoModel

@end
```

非常好，到这里，我们将不直接在ViewController中加载内容，相反，这部分逻辑将被抽象到另一个类中。创建一个名为`FRPPhotoImporter`的类。

到现在为止没有一处代码是关于函数式的。别担心，我们就要这么做了！这个`FRPPhotoImporter`将不会真正返回一个`FRPPhotoModel`对象，相反他会返回一些随身携带API最新的请求结果的信号。

```
@interface FRPPhotoImporter : NSObject
+ (RACSignal *)importPhotos;

@end
```

`FRPPhotoImporter`的`importPhotos`方法返回一个从API发送最新结果的RACSignal。这个RACSignal实际上是一个RACReplaySubject.但是由于ReactiveCocoa编程指南中不建议使用RACSubjects，我们申明的公共接口的返回类型为RACSignal而非RACSubject.现在让我们继续往下看:

```
+ (RACSignal *)importPhotos{
    RACReplaySubject * subject = [RACReplaySubject subject];
    NSURLRequest * request = [self popularURLRequest];
    [NSURLConnection sendAsynchronousRequest:request
                                       queue:[NSOperationQueue mainQueue]
                           completionHandler:^(NSURLResponse *response, NSData *data, NSErro
                               if (data) {
                                   id results = [NSJSONSerialization JSONObjectWithData:data

                                   [subject sendNext:[[[results[@"photos"] rac_sequence] map
                                       FRPPhotoModel * model = [FRPPhotoModel new];

                                       [self configurePhotoModel:model withDictionary:photoD
                                       [self downloadThumbnailForPhotoModel:model];

                                       return model;
                                   }] array]];

                                   [subject sendCompleted];
                               }
                               else{
                                   [subject sendError:connectionError];
```

```
                              }
    }];

    return subject;

}
```

这里面包含的内容太多，我们慢慢来整理一下：

- 首先我们创建了一个新的 `RACReplaySubject` 实例(这将是我们要返回的对象)。
- 其次我们创建了一个 `NSURLRequest` 来获取500px上热门的 `FRPPhotoModel` 数据。
- 随后我们发送一个网络的异步请求，并立即返回RACSubject对象。

这个直接返回的结果值得我们关注。

这个RACSubject对象被异步网络请求的回调block捕获，当API接口返回数据时回调block就会被调用，然后RACSubject对象会将结果传送出来，这些值将被我们的订阅了RACSubject信号的接收者所接受。

这是你看到的异步操作中，一个非常普通的模式。

1. 创建一个RACSubject.
2. 从异步调用的完成block中向RACSubject传送结果值。
3. 立即返回这个RACSubject对象

重要的是，要注意一个普通的RASSubject及其子类RACReplaySubject之间的区别。RACReplaySubject可以确保他背后的Subject只会被订阅一次，避免执行重复的操作(就像上面这种网络活动的情况)，RACReplaySubject将会缓存这个订阅的值，并将其转发给新的订阅者们--- 对我们的需求来说这非常完美。就像ReactiveCocoa的开发者Justin Spahr-Summers所指出的，这也能够避免可能的竞争状况。

我们发送了一个完整的数据集而不是单个随时间变化的流。如果我们连环地发送一个个单独的 `FRPPhotoModel` 流，这将'更加Reactive',也有助于实现分页的需求，但是我们不打算采用这种方式，因为他有点点'高级'了。你可以下载octokit：一个类似这种方式的例子。

URL请求的构造方法看起来应该是这样的:

```
+ (NSURLRequest *)popularURLRequest {
    return [AppDelegate.apiHelper urlRequestForPhotoFeature:PXAPIHelperPhotoFeaturePopula
                resultsPerPage:100 page:0
                photoSize:PXPhotoModelSizeThumbnail
                sortOrder:PXAPIHelperSortOrderRating
                except:PXPhotoModelCategoryNude];
}
```

subject发送什么，完全看不到好吗？呃。这取决于回调block.

```
if(data){
    id results = [NSJSONSerialization JSONObjectWithData:data options:0 error:nil];
    [subject sendNext:[[[results[@"photos"] rac_sequence] map:^id (NSDictionary *photoDic
```

```
        FRPPhotoModel *model = [FRPPhotoModel new];
        [self donwloadThumbnailForPhotoModel:model];

        return model;
    }] array]];

    [subject sendCompleted];
}
else{
    [subject sendError:connectionError];
}
```

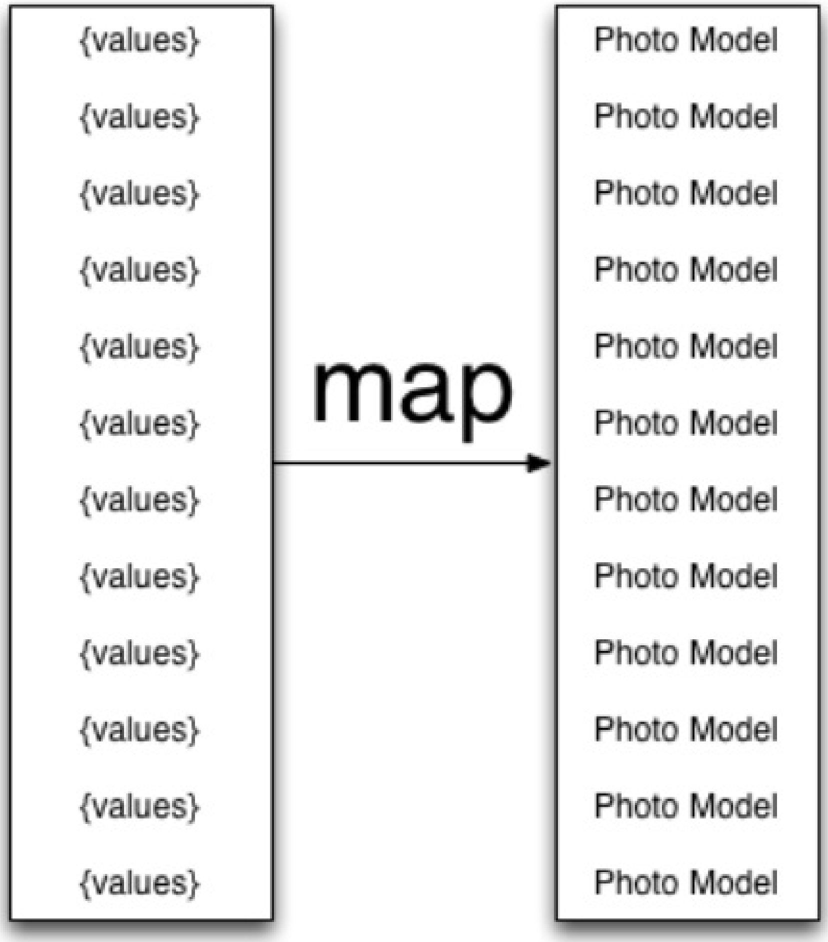
测试是否有数据返回时，可以说这不是一个很好的错误条件检测的方法，但这是一个教学的例子。如果数据为 `nil` ,我们会发送一个 `errorValue` ,否则我们会反序列化 `JSON` 数据并处理它。这不太容易很快就看清楚是怎么做到的，让我们来仔细看看。

```
[subject sendNext:[[[results[@"photos"] rac_sequence] map:^id (NSDictionary *photoDiction
    FRPPhotoModel * model = [FRPPhotoModel new];
    [self configurePhotoModel:model withDictionary:photoDictionary];
    [self downloadThumbnailForPhotoModel:model];
    return model;
}] array]];

[subject sendCompleted];
```

发送一个值，随着subject撸过去，第一个表达式结构相当简洁(但是场景很典型)。这个值是 `photos` 的值，然后转化为一个序列(sequence),然后做映射，最后转化为一个数组。这是上一章介绍的非常简单的 `map` 技术。

这个 `map` (映射)非常有意思。序列中的每一个元素，都会创建一个新的 `FRPPhotoModel` 对象、设置它然后返回它。为每一个 `results[ @"photos" ]` 的数组元素创建了一个 `FRPPhotoModel` 数组。这个数组就是随着subject发送过来的值。最后我们发送一个完成值 `completedValue` 好让订阅者们知道任务完成了。



注意在信号上手动附送值的能力是非典型的，这是RACSubject实例的专属能力。

`configurePhotoModel:withDictionary:` 方法，看起来应该像下面这样：

```
+ (void)configurePhotoModel:(FRPPhotoModel *)photomodel withDictionary:(NSDictionary *)di
    //Basic details fetched with the first, basic request
    photomodel.photoname = dictionary[@"name"];
    photomodel.identifier = dictionary[@"id"];
    photomodel.photographerName = dictionary[@"user"][@"username"];
    photomodel.rating = dictionary[@"rating"];

    photomodel.thumbnailURL = [self urlForImageSize:3 inArray:dictionary[@"images"]];

    //Extended attributes fetched with subsequent request
    if (dictionary[@"comments_count"]){
        photomodel.fullsizedURL = [self urlForImageSize:4 inArray:dictionary[@"images"]];
    }
}
```

除了URL的属性设置，都是最基本的东西。依靠其他的方法来从500px的API中返回的图片列表中提取正确的url信息。500px API返回的数据结构是下面这样的格式：

```
(
    {
```

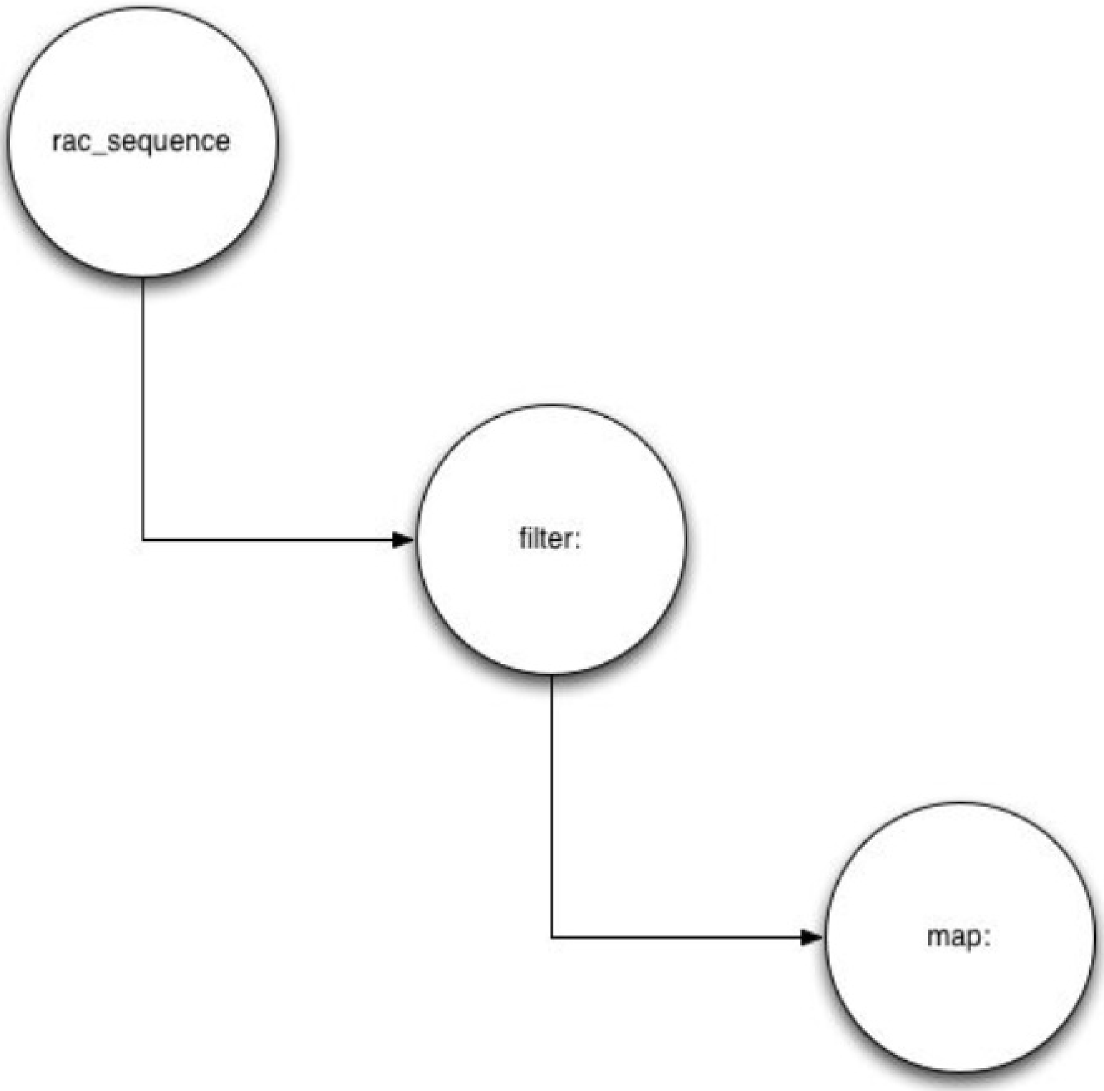
```
        size = size;
        url = ...;
    }
)
```

这是一个字典数组，每一个字典中包含一个`size`字段和一个`url`字段。我们读取这样字段的方法如下：

```
+ (NSString *)urlForImageSize:(NSInteger)size inDictionary:(NSArray *)array{
    return [[[[[array rac_sequence] filter:^ BOOL (NSDictionary * value){
        return [value[@"size"] integerValue] == size;
    }] map:^id (id value){
        return value[@"url"];
    }] array] firstObject];
}
```

这里有一些隐含的错误处理，如果序列为空， `NSArray`的`firstObject`方法默认返回nil.

- 第一步，我们过滤掉那些`size`字段不匹配要求的字典。
- 然后，将这些符合要求的字典做一次映射来提取字典中`url`字段的内容。
- 最后，我们获得一个NSString 对象的序列，把它转化为数组，然后返回`firstObject`.



在ReactiveCocoa中类似上面的链式调用非常常见。值从 `rac_sequence` 推送到 `filter:` 方法中，最后推送到 `map:` 方法里。最后调用序列 `rac_sequence` 的 `array` 方法，将序列的结果转化为 `array` .

最后，我们的 `downloadThumbnailForPhotoModel:` 方法，看起来应该是下面这样：

```
+ (void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel{
    NSAssert(photoModel.thumbnailURL, @"Thumbnail URL must not be nil");

    NSURLRequest * request = [NSURLRequest requestWithURL:[NSURL URLWithString:photoModel
    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:^(NSURLResponse *response, NSData *data, NSError * connectionEr
            photoModel.thumbnailData = data;
    }];
}
```

这个方法里面没有任何的关于 `Reactive` 的部分---仅仅是下载thumbnail的url，然后在完成块中适当地设置相关属性。

我们几乎做完了这个画廊所需要的所有基础的事情，接下来，我们看看 `viewController` .在实现文件里定义下面的的私有属性。

```
@interface FRPGalleryViewController ()
@property (nonatomic , strong) NSArray *photoArray;

@end
```

来看下viewDidLoad中的实现。

```
static NSString * CellIdentifier = @"Cell";

- (void)viewDidLoad{
    [super ViewDidLoad];

    //Configure self
    self.title = @"Popular on 500px";

    //Configure View
    [self.collectionView registerClass:[FRPCell class] forCellWithReuseIdentifier:CellIde

    //Reactive Stuff
    @weakify(self);
    [RACObserver(self, photosArray) subscribeNext:^(id x){
        @strongify(self);
        [self.collectionView reloadData];
    }];

    //Load data
    [self loadPopularPhotos];
}
```

我们为viewController设置了一个title并且为collectionView注册了一个类，collectionView将会在他的cells中复用这个类的实例。这里我引用了一个不存在的UICollectionViewCell的子类，我们很快会创建她。

在'Reactive Stuff'注释之下，你会发现一些奇怪的语法。

```
@weakify(self);
[RACObserver(self, photosArray) subscribeNext:^(id x){
    @strongify(self);
    [self.collectionView reloadData];
}];
```

`RACObserver` 是一个C的宏定义，带两个参数：对象及对象某个属性的 `keyPath` （关键路径）。他会返回一个带属性值的信号,无论这个属性的值怎么变都会及时地通过该信号反馈出来。在这里当self结束分配的时候会发送一个 `completion Value` 的值。订阅这个信号的目的是无论我们的photosArray中的元素属性怎么变，我们都能够在collectionView重新加载的时候实时获取反馈。

在Objective-C的ARC条件下@weakify/@strongify这个双人舞是非常常见的。@weakify创建一个新的self的

弱引用weakself，@strongify创建这个weakself的强引用，并在@strongify的作用域中起作用。strongify的这种做法，一般称为“影子变量”，那是因为这个新的强引用的变量就叫`self`,替代了原本强引用的self.

一般而言，`subscribeNext:`的block将捕获其词法范围内的self，造成self和block之间的循环引用。block被`subscribeNext:`的返回值，一个RACSubscriber实例，强引用，然后被RACObserver宏捕获。解除分配时，RACOberver会自动解除第一个参数的分配，这样的话self就应该被解除分配，但self被block强引用，self要得以解除分配的唯一条件即引用计数为0，这样的话就必须先解除block的分配，而前面的分析我们知道block被RACSubscriber实例引用，而该实例默认被self强引用，因此，如果不调用weakify/strongify，self就永远也不可能解除分配。

最后，我们实际来调用`loadPopularPhotos`(他的实现如下)

```
- (void)loadPopularPhotos{
    [[FRPPhotoImporter importPhotos] subscribeNext:^(id x){
        self.photosArray = x;
    } error:^(NSError * error){
        NSLog(@"Couldn't fetch photofrom 500px: %@",error);
    }];
}
```

这个方法实际上负责调用`FRPPhotoImporter`的`importPhotos`方法（现在请加上他的头文件），他订阅了我们私有成员属性的结果。由于UICollectionViewDataSource协议的架构，我们不得不把这些状态引入进来。

现在让我们来看一下这些协议方法，有两个是必须的，实现如下：

```
- (NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NS
    return self.photosArray.count;
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellForItemAt
    FRPCell * cell = [collectionView dequeueReusableCellWithReuseIdentifier:CellIdentifie
    [cell setPhotoModel:self.photosArray[indexPath.row]];

    return cell;
}
```

第一个方法简单地返回了collectionView中的cell的数量，在这里，准确地讲是photosArray属性的cell数量。接下来的这个方法从collectionView列表中获得了一个cell实例，并调用其上的`setPhotoModel:`方法（这个我们还没有实现，但别担心）。这些代码应该看起来非常熟悉，如果你曾经处理过UITableViewDataSource的方法的话。

这就是我们`ViewController`完整的实现。现在我们来创建UICollectionViewCell的子类，命名为`FRPCell`，像下面这样来修改他的头文件。

```
@class FRPPhotoModel;

@interface FRPCell : UICollectionViewCell
```

```
- (void)setPhotoModel:(FRPPhotoModel *)photoModel;
@end
```

在实现文件中添加下面的私有扩展：

```
#import "FRPPhotoModel.h"
@interface FRPCell ()
@property (nonatomic , weak ) UIImageView * imageView;
@property (nonatomic , strong ) RACDisposeable *subscription;

@end
```

这里有两个属性：一个图片视图和一个订阅者。图片视图是弱引用，因为它属于父视图（这是UICollectionViewCell的一个标准的用法），我们将实例化并赋值给imageView。接下来的属性是一个订阅，当使用ReactiveCocoa来设置图像视图的图像属性时，我们将接触到它。注意它必须是强引用而非弱引用否则你会得到一个运行时的异常。

```
- (id)initWithFrame:(CGRect)frame{
    self = [super initWithFrame:frame];
    if(!self) return nil;

    //Configure self
    self.backgroundColor = []UIColor darkGrayColor];

    //Configure subviews
    UIImageView * imageView = [[UIImageView alloc] initWithFrame:self.bounds];
    imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFle
    [self.contentView addsubView:imageView];
    self.imageView = imageView;

    return self;
}
```

标准的UICollectionView子类的模版会创建并分配imageView属性。注意，我们必须有一个（被self）强引用的本地变量作为中介来存储imageView，这样就不会在赋值给self的imageView属性的时候，imageView被立即解除分配。否则会有编译错误。

完成我们的500px画廊，我们还需要实现两个方法，第一个就是 `setPhotoModel:` 方法

```
- (void)setPhotoModel:(FRPPhotoModel *)photoModel{
    self.subscription = [[[RACObserver(photoModel, thumbnailData)
        filter:^ BOOL (id value){
            return value != nil;
        }] map:^id (id value){
            return [UIImage imageWithData:value];
        }] setKeyPath:@keypath(self.imageView, image) onObject:self.imageView];
}
```

这种方法来给订阅的属性赋值，我们老早就知道了。它把 `setKeyPath:OnObject:` 的返回值赋给

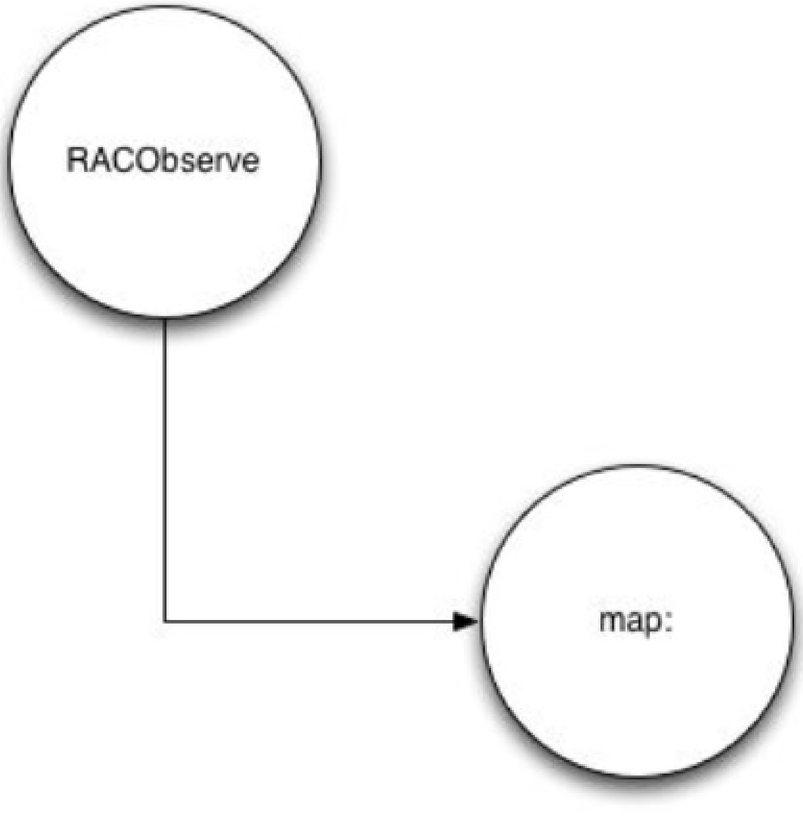
了`self.subscription`.实践中这种方法根本不使用，我们使用RAC的C语法宏来代替，不久之后我们就会涉及这方面的知识。

两个原因导致订阅是必要的：

```
1. 当它没有接受一个新的值时，我们想延迟处理。
2. 信号的订阅通常是冷信号，除非有人订阅他（信号），否则信号不会起作用。
```

`setKeyPath:onObject:`是`RACSignal`的一个方法：绑定最新的信号的值给对象的关键路径。在这里我们在一个级联的信号上调用了这个方法，让我们来仔细看看：

```
[[RACObserver (photoModel, thumbnailData)
    filter:^BOOL (id value){
        return value != nil;
    }] map:^ id (id value){
        return [UIImage imageWithData:value];
    }];
```



信号由`RACObserver`这个C的宏生成，这个宏简单地返回一个监控目标对象关键路径值变化的信号。在我们这个例子中，我们的目标对象是`photoModel`，关键路径为`thumbnailData`属性。我们过滤掉所有的nil值，然后对过滤后的值做映射：把NSData实例转为UIImage对象。

注意，把NSData实例转化为UIImage的这个映射仅在小图上可以很好地运行，如果频繁地做这个映射或者作用到大图上会引起性能问题。理想的情况下，我们会缓存这些已经解压的图像以避免每一次都重复计算。这个技术不是本书所讨论的范畴，但我们将使用另一个通过ReactiveCocoa来实现的方法。

thumbnailData属性根本不需要在这里设置，他可以在稍后的某个时间在应用的其他部分来完成设置，然后cell的图像就会像魔术一般更新。

可以让我们稍微突破一下Model-View-Controller模式好吗？只是一点点的不守规矩。幸运的是，下一章我们将看到无处不在的MVC模式的困境，所以我们不必担心这一点点的突破，一点点的改进。

上面提到的 `setKeyPath:onObject:` 方法中，一旦 `onObject:` 对象被释放，他的订阅也会被自动取消。我们的cell实例是被collectionView所复用的，因此在复用的时候，我们需要取消cell上各组件的订阅。我们可以通过重写 `UICollectionViewCell` 的下列方法达成：

```
- (void)perpareForReuse {
    [super prepareForReuse];

    [self.subscription dispose], self.subscription = nil;
}
```

这个方法在Cell被复用之前调用。如果现在运行我的应用，我们可以看到下面的结果：


Carrier
7:06 PM
Popular on 500px

太好了！我们可以通过滚动视图来证实我们手动处理订阅的有效性。

添加FunctionalReactivePixels

一个简单的画廊弄好了，但是我们是不是想看一下高清图呢？当用户点击画廊中的某一个单元格时，我们创建一个新的视图控制器并将其推入到导航堆栈中。

```
- (void)collectionView:(UICollectionView *)collectionView
    didSelectItemAtIndexPath:(NSIndexPath *)indexPath{
    FRPFullSizePhotoViewController * viewController = [[FRPFullSizePhotoViewController al

    viewController.delegate = self;
    [self.navigationController pushViewController:viewController animated:YES];

}
```

这个方法没有任何特殊的，只是些一般的OC方法。当然别忘了在当前实现文件里加载视图控制器(FRPFullSizePhotoViewControler)的头文件.现在让我们来创建这个视图控制器(FRPFullSizePhotoViewControler).

创建一个UIViewController的子类FRPFullSizePhotoViewControler,这不会是一个特别的'Reactive'的视图控制器，实际上大部分只是 `UIPageViewController` 子视图控制器的模版。

```
@class FRPFullSizePhotoViewController;

@protocol FRPFullSizePhotoViewControllerDelegate <NSOject>
- (void)userDidScroll:(FRPFullSizePhotoViewController *)viewController toPhotoAtIndex:(NS

@end

@interface FRPFullSizePhotoViewController : UIViewController

- (instancetype)initWithPhotoModels:(NSArray *)photoModelArray currentPhotoIndex:(NSInteg

@property (nonatomic , readonly) NSArray *photoModelArray;
@property (nonatomic, weak) id<FRPFullSizePhotoViewControllerDelegate> delegate;

@end
```

回到画廊视图控制器实现必要的代理方法：

```
- (void)userDidScroll:(FRPFullSizePhotoViewController *)viewController toPhotoAtIndex:(NS
    [self.collectionView scrollToItemAtIndexPath:[NSIndexPath indexPathForItem:index inSe
                atScrollPosition:UICollectionViewScrollPositionCenteredVertically
                        animated:NO];
}
```

当我们滑到一个新的图像去查看其高清图片时，这个方法将更新collectionView滑动的位置。这样一来，当

用户查看完高清图回到这个界面的时候，高清图所对应的缩略图将会显示在界面上，方便用户获知自己浏览的位置以及继续往下浏览。

`#import` 这些必要的数据模型的头文件并追加一下两个私有属性：

```
@interface FRPFullSizePhotoViewController () <UIPageViewControllerDataSource, UIPageViewC
//Private assignment
@property (nonatomic, strong) NSArray *photoModeArray;
//Private properties
@property (nonatomic, strong) UIPageViewController *pageViewController;
@end
```

`photoModelArray` 是共有的只读属性，但是内部可读写。第二个属性是我们的子视图控制器。我们这样来初始化：

```
- (instancetype)initWithPhotoModels:(NSArray *)photoModelArray currentPhotoIndex:(NSInteg
    self = [self init];
    if (!self) return nil;

    //Initialized, read-only properties
    self.photoModelArray = photoModelArray;

    //Configure self
    self.title = [self.photoModelArray[photoIndex] photoName];

    //ViewControllers
    self.pageViewController = [UIPageViewController alloc]
                                initWithTransitionStyle:UIPageViewControlerTransition
                                navigationOrientation:UIPageViewControllerNavigationO
                                options:@{ UIPageViewControllerInterPageSpacingKey: @
    self.pageViewController.dataSource = self;
    self.pageViewController.delegate = self;
    [self addchildViewController:self.pageViewController];

    [self.pageViewController setViewController:@[[self photoViewControllerForIndex:photoI
                        direction:UIPageViewControllerNavigationDirectionForward
                        animated:NO completion:nil ];

    return self;
}
```

赋值属性、设置标题、配置我们的 `pageViewController` ，一切都非常无聊，我们的viewDidLoad方法也同样简单。

```
- (void)viewDidLoad{
    [super viewDidLoad];

    self.view,backGroundColor = [UIColor blackColor];

    self.pageViewController.view.frame = self.view.bounds;
```

```
    [self.view addSubView:self.pageViewController.view];
}
```

我要指出的是，简便起见，在我的应用里我禁用了横向展示，因为这不是一本关于`autoresizingMask`或者`autoLayout`的书。你可以通过Eria Sadun的书了解更多关于`autoLayout`方面的细节。

下面我们来了解一下UIPageViewController的数据源协议和代理协议。

```
- (void)pageViewController:(UIPageViewController *)pageViewController
    didFinishAnimating: (BOOL)finished
    previousViewControllers:(NSArray *)previousViewControllers
    transitionCompleted:(BOOL)completed{
        self.title = [[self.pageViewController.viewControllers.firstObject photoModel] ph
        [self.delegate userDidScroll:self toPhotoAtIndex:[self.pageViewController.viewCon
    }

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController viewC
    return [self photoViewControllerForIndex:viewController.photoIndex - 1];
}

- (UIViewController *)pageViewController:(UIPageViewController *)pageViewController viewC
    return [self photoViewControllerForIndex:viewController.photoIndex + 1];
}
```

虽然这些方法没有技术上的`reactive`，却体现出一定意义上的实用性。我很佩服这种在特殊类型的视图控制器上的抽像，干得漂亮，Apple！

我们的视图控制器创建方法，类似下面这样：

```
- (FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index{
    if (index >= 0 && index < self.photoModelArray.count){
        FRPPhotoModel *photoModel = self.photoModelArray[index];

        FRPPhotoViewController *photoViewController = [[FRPPhotoViewController alloc] ini

        return photoViewController;
    }

    //Index was out of bounds, return nil
    return nil;
 }
```

它基本上创建比配置了一个我们将要使用的UIViewController的子视图控制器FRPPhotoViewController。下面是他的头文件：

```
@class FRPPhotoModel;

@interface FRPPhotoViewController : UIViewController
- (instancetype)initWithPhotoModel:(FRPPhotoModel *)photoModel index:(NSInteger)photoInde
```

```
@property (nonatomic, readonly) NSInteger photoIndex;
@property (nonatomic, readonly) FRPPhotoModel * photoModel;

@end
```

这个视图控制器非常简单：显示一个photoModel下的高清图片，并提示photoImporter(单例对象)下载这个图片。它是如此简单，我现在就告诉你它的全部实现。

```
//Model
#import "FRPPhotoModel.h"

//Utilities
#import "FRPPhotoImporter.h"
#import <SVProgressHUD.h>

@interface FRPPhotoViewController ()
//Private assignment
@property (nonatomic, assign) NSInteger photoIndex;
@property (nonatomic, strong) FRPPhotoModel *photoModel;

//Private properties
@property (nonatomic, weak) UIImageView * imageView;

@end

@implementation FRPPhotoViewController

- (instancetype)initWithPhotoModel:(FRPPhotoModel *)photoModel index:(NSInteger)photoInde
    self = [self init];
    if (!self) return nil;

    self.photoModel = photoModel;
    self.photoIndex = photoIndex;

    return self;
}

- (void)viewDidLoad{
    [super viewDidLoad];

    //Configure self's view
    self.view.backGroundColor = [UIColor blackColor];

    //Configure subViews
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:self.view.bounds];

    RAC(imageView, image) = [RACObserve(self.photoModel, fullsizeData) map:^id (id value)
                                    return [UIImage imageWithData:value];
                                }];

    imageView.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubView:imageView];
    self.imageView = imageView;
}

- (void)viewWillAppear:(BOOL)animated{
```

```
    [super viewWillAppear:animated];
    [SVProgressHUD show];

    //Fetch data
    [[FRPPhotoImporter fetchPhotoDetails:self.photoModel]
            subscribeError:^(NSError *error){
                [SVProgressHUD showErrorWithStatus:@"Error"];
            }
            completed:^{
                [SVProgressHUD dismiss];
            }];
}

@end
```

就像我们的collectionViewCell中那样，我们将UIImageView的image属性和数据模型的某个属性映射后的值绑定，所不同的是ViewController不需要考虑复用，所以我们不必计较怎么取消imageView的订阅---当imageView对象解除分配的时候，订阅将会被取消。

这个实现里面另一个有趣的部分在 `viewWillAppear:` 里：

```
[SVProgressHUD show];
//Fetch data
[[FRPPhotoImporter fetchPhotoDetails:self.photoModel]
        subscribeError:^(NSError * error){
            [SVProgressHUD showErrorWithStatus:@"Error"];
        }
        completed:^{
            [SVProgressHUD dismiss];
        }];
```

没有收到错误或者完成信息之前，我们必须给用户展示网络请求的状态。你看，500px的受欢迎的照片的API接口只返回了一个照片的大概信息，但我们需要这个照片更详细的信息，所以我们必须调用第二个API接口来获取每一个照片的详细信息（包括全尺寸照片的URL）。

```
+ (NSURLRequest *)photoURLRequest:(FRPPhotoModel *)photoModel{
    return [AppDelegate.apiHelper urlRequestForPhotoID:photoModel.identifier.integerValue
}
```

我们还没有实现 `fetchPhotoDetails:` 方法，所以现在我们回到 `FRPPhotoImporter` 中，在头文件中定义这个方法，在实现文件中实现它。

```
+ (RACReplaySubject *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    RACReplaySubject * subject = [RACReplaySubject subject];
    NSURLRequest *request = [self photoURLRequest:photoModel];

    [NSURLConnection sendAsynchronousRequest:request
        queue:[NSOperationQueue mainQueue]
        completionHandler:^ (NSURLResponse *response, NSData * data, NSError *connectionE
```

```
            if(data){
                id results = [NSJSONSerialization JSONObjectWithData:data options:0 error

                [self configurePhotoModel:photoModel withDictionary:results];
                [self downloadFullsizedImageForPhotoModel:photoModel];

                [subject sendNext:photoModel];
                [subject sendCompleted];
            }
            else{
                [subject sendError:connectionError];
            }
        }];

    return subject;
}
```

这种方法跟前面我们看到的`importPhotos`方法模式一样，我们的`downloadFullsizedImageForPhotoModel:`方法跟`downloadThumbnailForPhotoModel:`方法也是一样的。除了这两者之外，还有什么重要的抽象方法呢？让我们来完成我们的缩略图方法。

```
+ (void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.thumbnailURL withCompletion:^(NSData *data){
        photoModel.thumbnailData = data;
    }];
}

+ (void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.fullsizedURL withCompletion:^(NSData * data){
        photoModel.fullsizedData = data;
    }];
}

+ (void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.fullsizedURL withCompletion:^(NSData *data){
        photoModel.fullsizedData = data;
    }];
}

+ (void)download:(NSString *)urlString withCompletion:(void(^)(NSData * data))completion{
    NSAssert(urlString, @"URL must not be nil" );

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:urlString]]
    [NSURLConnnection sendAsynchronousRequest:request queue:[NSOperationQueue mainQueue]
        if (completion){
            completion(data);
        }
    }];
}
```

我曾经与这样一位客户工作过，他认为如果你某行一样的代码重复写两次，这代码就应该得到某种程度的抽象。虽然我认为这有点偏激，但我喜欢这种态度。

好了。我们现在可以运行这个应用，点击一个图片去查看它的高清图片。我们也可以向前或者向后滑动来查看前一个或后一个高清图片。非常棒！


Carrier
7:38 PM
Back
Lightning ...

和FunctionalReactivePixels一起实践

上一节，我们很多次使用了 `ReactiveCocoa` 的关键部分，这里有更多的机会来使用 `ReactiveCocoa` 整个代码库。开始吧！

首先在我们的画廊视图控制器中实现三个不同的代理方法：`CollectionViewDataSource`、`CollectionViewDelegate`、高清图视图控制器的 `PhotoViewControllerDelegate`

使用一个称之为 `RACDelegateProxy` 的实例，我们可以抽象委托类型的协议的任何方法实现(比如：那些返回void类型的)。

委托代理是一个称为 `rac_signalForSelector:` 对象的'白板'，获取当Selector被调用时发送的新值的信号。

注意：你必须retain这个delegate对象，否则他们将会被释放，你将会得到一个 `EXC_BAD_ACCESS` 异常。添加下列私有属性到画廊视图控制器：

```
@property (nonatomic, strong) id collectionViewDelegate;
```

同时你也需要导入 `RACDelegateProxy.h`，因为他不是ReactiveCocoa的核心部分，不包含在 `ReactiveCocoa.h` 中。移除 `UICollectionViewDelegate` 以及 `FRPFullsizePhotoViewControllerDelegate` 方法，追加下面的代码到 `viewDidLoad`.

```
RACDelegateProxy *viewControllerDelegate = [[RACDelegateProxy alloc]
                                    initWithProtocol:@protocol(FRPFullSizePhotoViewContro

[[viewControllerDelegate rac_signalForSelector:@selector(userDidScroll:toPhotoAtIndex:)
        subscribeNext:^(RACTuple *value){
            @strongify(self);
            [self.collectionView
                scrollToItemAtIndexPath:[NSIndexPath indexPathForItem:[value.second integ
                atScrollPosition:UICollectionViewScrollPositionCenteredVertically
                animated:NO];
        }];

self.collectionViewDelegate = [[RACDelegateProxy alloc] initWithProtocol:@protocol(UIColl

[[self.collectionViewDelegate rac_signalForSelector:@selector(collectionView:didSelectIte
        subscribeNext:^(RACTuple *arguments) {
            @strongify(self);
            FRPFullSizePhotoViewController *viewController = [[FRPFullSizePhotoViewContro
            viewController.delegate = (id<FRPFullSizePhotoViewControllerDelegate>)viewCon

            [self.navigationController pushViewController:viewController animated:YES];

        }];
```

我们也可以在`self`上调用`rac_signalForSelector:`，使用同样的block块。然而，我们有必要在视图控制器实现里提供一个空存根方法以避免编译器发出"实现不完全"之类的警告。

> 空存根方法：源于C++的一个非常不错的函数设计方法。在设计整个程序时，一般会先编写完所有的代码，然后开始编译和测试，但这样有时候会出现一大堆错误而不知从哪里入手，这时我们可以采用空存根技术。
>
> 存根是一个仅仅返回某个意义不大的值的空函数。存根可以用来测试整个程序的逻辑关系，以及分块实现程序的不同部分。
>
> 设计一个程序时，先分析设计程序的各个函数完成的功能；然后直接设计函数的存根并编译，编译通过，证明程序的逻辑关系没有问题的情况下，再来分别实现各个不同的函数(存根)。

接下来，我们有更多的机会来抽象这个类中的方法。`loadPopularPhotos`方法除了改变我们的状态之外，并没有什么卵用。如果`ReactiveCocoa`能够很好地监控这些状态，让我们不在这方面担心的话，那肯定是极好的！幸运的是，我恰好知道这个~

我们移除这个方法，在viewDidLoad中键入下面的代码来代码这个方法的调用：

```
RACSignal *photoSignal = [FRPPhotoImporter importPhotos];
RACSignal *photosLoaded = [photoSignal catch:^RACSignal *(NSError *error) {
    NSLog(@"Couldn't fetch photos from 500px : %@",error);
    return [RACSignal empty];
}];
RAC(self, photosArray) = photosLoaded;
[photosLoaded subscribeCompleted: ^{
    @strongify(self);
    [self.collectionView reloadData];
}];
```

一开始我们只是进行了`importPhotos`方法调用，不同的是，我们用`signal`来存放其返回值。 然后，我们"捕抓"这个信号上的错误并将它打印出来(跟我们之前做的一样，只不过语法不同而已)。比起`subscribeError:`方法，`catch:`方法处理的更为巧妙：它允许无错误值的信号穿透它，仅在信号有错误事件发生时才会调用它的block并发送其在发生错误时的返回值。这里我们使用`catch:`方法，来过滤无错误的值。这个`catch:`块仅仅返回一个空信号。更多关于这方面知识的细节请参考StackOverFlow的问题。

上面的方式，有一点点污染了我们的局部变量作用域，这可以用下面的更简洁的等效方法：

```
RAC(self, photosArray) = [[[[FRPPhotoImporter importPhotos]
        doCompleted:^{
            @strongify(self);
            [self.collectionView reloadData];
        }] logError] catchTo:[RACSignal empty]];
```

使用RAC宏，我们创建了`photosLoaded`信号的最新值到`photoArray`属性的单向绑定。太好了，保持状态!

我们来看一下，我们的collectionViewCell的子类实现：

```
@interface FRPCell ()

@property (nonatomic, weak) UIImageView *imageView;
@property (nonatomic, strong) RACDisposable *subscription;

@end

@implementation FRPCell

- (instancetype)initWithFrame:(CGRect)frame {
    ...
}

- (void)perpareForReuse {
    [super perpareForReuse];

    [self.subscription dispose], self.subscription = nil;
}

- (void)setPhotoModel:(FRPPhotoModel *)photoModel {
    self.subscription = [[[RACObserve(photoModel, thumbnailData) filter:^BOOL(id value) {
        return value != nil;
    }] map:^id(id value) {
        return [UIImage imageWithData:value];
    }] setKeyPath:@keypath(self.imageView, image) onObject:self.imageView];
}

@end
```

这里有两个标志性的点表明了一个使用ReactiveCocoa来抽象的机会。

1. 我们有状态(`subscription` 属性)
2. 我们手动处理 `RACDisposable` 的生命周期

无论何时调用一个 `RACDisposable` 对象的 `dispose` 方法，就是一个"这里有更加响应式的方法来作某件事"的好信号。在我们的例子中，这种嗅觉是对的。

通过在 `FRPCell` 创建一个新的属性，我们能够抽象掉使用 `prepareForReuse` 方法的必要性。这个属性就是 `photoModel` (我们之前的行为就像是一个只写的属性，现在它将变为可读写的了)。把属性放在文件顶部：

```
@property (nonatomic, strong ) FRPPhotoModel *photoModel;
```

下一步我们将彻底摆脱 `setPhotoModel:` 方法。我们将为 `photoModel的thumbnailData` 观察我们自己的关键路径。将下面的代码添加到cell的初始化函数中。

```
RAC(self.imageView, image) = [[RACObserve(self, photoModel.thumbnailData) ignore:nil]
                            map:^(NSData *data){
                                return [UIImage imageWithData:data];
                            }];
```

注意看我们观察的是`self`的`photoModel.thumbnailData`的关键路径，而非`self.photoModel`的`thumbnailData`的关键路径。这点微妙的区别，作用却大大不同。当`self`的属性`photoModel`或者`photoModel`的`thumbnailData`属性改变时，关键路径`photoModel.thumbnailData`将会收到一个被(这种变化所)引发的KVO消息。

现在我们总算彻底摆脱了`subscription`属性！

网络层回访

还有一个机会来进一步接受我们函数反应型编程的理念，那就是我们的网络层 `FRPPhotoImporter`,我们先来看看下载图片的方法：

```
+ (void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.thumbnailURL withCompletion:^(NSData *data) {
        photoModel.thumbnailData = data;
    }];
}

+ (void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    [self download:photoModel.fullsizedURL withCompletion:^(NSData *data){
        photoModel.fullsizedData = data;
    }];
}

+ (void)download:(NSString *)urlString withCompletion:(void (^)(NSData *data))completion
    NSAssert(urlString, @"URL must not be nil");

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString:urlString]]
    [NSURLConnection sendAsynchronousRequest:request
                                queue:[NSOperationQueue mainQueue]
                                completionHandler:
                                    ^(NSURLResponse *response, NSData *data, NSError *co
                                           if(completion) {
                                               completion(data);
                                           }
                                    }];
}
```

Completion blocks?这是另外一个使用Signals的机会。更深入一点来说，我们可以使用 `NSURLConnection` 的ReactiveCocoa的扩展。下面我们来重写上面的方法：

```
+ (void)downloadThumbnailForPhotoModel:(FRPPhotoModel *)photoModel {
    RAC(photoModel, thumbnailData) = [self download:photoModel.thumbnailURL];
}

+ (void)downloadFullsizedImageForPhotoModel:(FRPPhotoModel *)photoModel {
    RAC(photoModel,fullsizedData) = [self download:photoModel.fullsizedURL];
}

+ (RACSignal *)download:(NSString *)urlString {
    NSAssert(urlString , @"URL must not be nil");

    NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL URLWithString: urlString]

    return [[[NSURLConnection rac_sendAsynchronousRequest:request]
                map:^id (RACTuple *value) {
                    return [value second];
                }] deliverOn:[RACScheduler mainThreadScheduler]];
}
```

这里有两个大的不同：

1. 我们使用RAC来绑定`downloadFullsizedImageForPhotoModel:`返回的信号的最新值。
2. 我们返回`NSURLConnection的rac_sendAsynchronousRequest:`返回值的映射。

我们来看看这里究竟发生了什么。 看文档：`rac_sendAsynchronousRequest:`返回一个发送网络请求响应值的信号。 `RACTuple`它所发送的内容分别包含响应和数据。有网络错误发生时，它会抛出错误。 最后我们改变线程的调度，将signal切换到主线程上。 (一个线程的调度者类似于一个线程。)

看，网络信号将会把它的值返回给后台的调度者，如果我们不阻止它，它可能最终会去从事更新UI的事件，而后台线程是没有能力更新UI的。

我们回过头来看看最开始的那两行。注意下这行：

```
RAC(photoModel, thumbnailData) = [self download:photoModel.thumbnailURL];
```

通常，我不推荐将一个model绑定到多个signal，然而，我们知道这个信号会在完成网络调用后立即执行完并结束订阅。只要我们仅在一个实例上绑定这个keyPath，这种就是安全的。

我们可以用类似的方式抽象掉使用`RACReplaySubject`的部分,来重新审视我们的`fetchPhotoDetails:`方法吧。

```
+ (RACReplaySubject *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    RACReplaySubject *subject = [RACReplaySubject subject];

    NSURLRequest *request = [self photoURLRequest:photoModel];
    [NSURLConnection sendAsynchronousRequest:request
                 queue:[NSOperationQueue mainQueue]
    completionHandler:^(NSURLResponse *response, NSData *data, NSError *connectionError)
            if(data) {
                id results = [NSJSONSerialization JSONObjectWithData:data options:0 error
                [self configurePhotoModel:photoModel withDictionay:results];
                [self downloadFullsizedImageForPhotoModel:photoModel];
                [subject sendNext:photoModel];
                [subject sendCompleted];
            }
            else {
                [subject sendError:connectionError];
            }
    }];

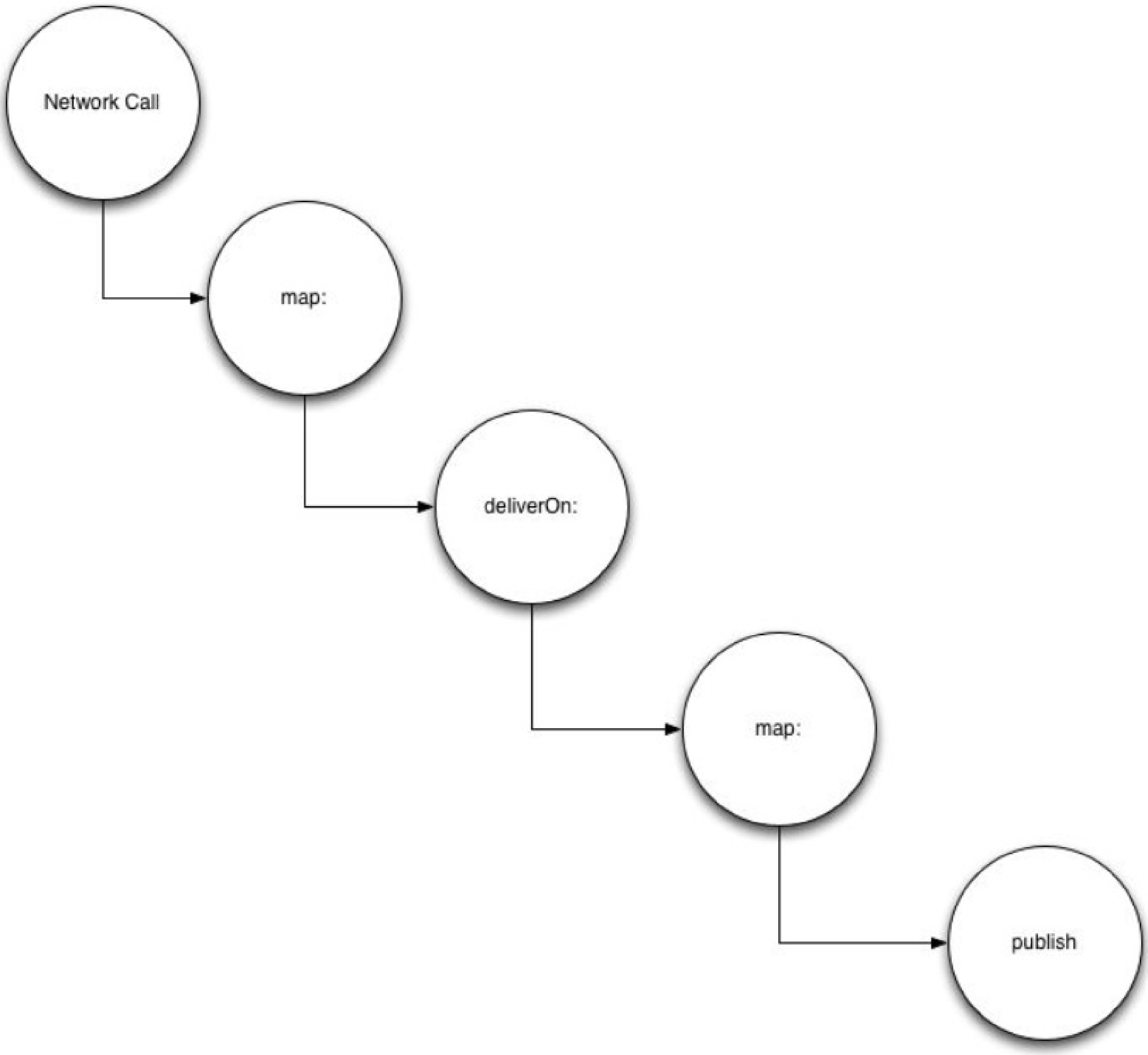

    return subject;

}
```

有一点点凌乱，我们来整理下。

```
+ (RACSignal *)fetchPhotoDetails:(FRPPhotoModel *)photoModel {
    NSURLRequest *request = [self photoURLRequest:photoModel];
    return [[[[[[NSURLConnection rac_sendAsynchronousRequest:request]
                                map:^id(RACTuple *value){
                                    return [value second];
                                }]
                                deliverOn:[RACScheduler mainThreadScheduler]]
                                    map:^id (NSData *data) {
                                        id results = [NSJSONSerialization JSONObjectWithD
                                                               options:0 error:ni
                                        [self configurePhotoModel:photoModel withDictiona
                                        [self downloadFullsizedImageForPhotoModel:photoMo
                                        return photoModel;
                                    }] publish] autoconnect];
}
```

注意： 返回值从 `RACReplaySubject *` 变成了 `RACSignal *` . 这里有很多地方需要梳理，所以我们提前做了下面这个示意图来说明：



我们已经知道 `deliverOn:` 是怎样工作的，所以让我们来关注信号链条最末端的信号操作 `publish` . `publish` 返回一个 `RACMulitcastConnection` ,当信号连接上时，他将订阅该接收信号。 `autoconnect` 为

我们做的是：当它返回的信号被订阅，连接到 该(订阅背后的)信号（underly signal）。

执行获取每一个订阅，在订阅的时候，我们返回的信号将会变“冷”。那是因为我们对底层信号进行多播，网络请求只会执行一次，但是它的结果被多播。这会导致：网络信号将只会被执行一次（当它被订阅时执行），是冷的(直到订阅为止，它不会被执行)，甚至可删除的(如果一次性处理订阅的生成)。

基本上，我们能保证信号只会被订阅一次，我们不需要回滚(replay).

注意：我们可以用下面的 `reduceEach:` 替代使用 `RACTuple` 的第一个 `map:` ，以便提供编译时检查。

```
reduceEach:^id(NSURLResponse *response, NSData *data) {
    return data;
}]
```

剩下的网络访问接口， `importPhotos` 方法重构如下：

```
+ (RACSignal *)importPhotos {
    NSURLRequest *request = [self popularURLRequest];

    return [[[[[[NSURLConnection rac_sendAsynchronousRequest:request]
                reduceEach:^id(NSURLResponse *response , NSData *data){
                    return data;
                }]
                deliverOn:[RACScheduler mainThreadScheduler]]
                map:^id (NSData *data) {
                    id results = [NSJSONSerialization JSONObjectWithData:data options:0 e
                    return [[[results[@"photo"] rac_sequence]
                        map:^id (NSDictionary *photoDictionary) {
                            FRPPhotoModel *model = [FRPPhotoModel new];
                            [self configurePhotoModel:model withDictionary:photoDictionar
                            [self downloadThumbnailForPhotoModel:model];
                            return model;
                        }] array];
                }] publish] autoconnect];
}
```

总结

本章我们使用 `ReactiveCocoa` 做了很多实践，总结了几个关键点：

- 函数式编程可在任何地方起作用
    - 数据导入的代码，即使没有反应式代码，我们也能够使用 `map:` 和 `filter:` 来帮忙。在抽象方面，总觉得从未被实际实现。

- 为函数的副作用使用 `subscribeNext:`
    - `subscribeNext:` 和其他类似的方法订阅信号的副作用，返回 `RACDisposable` 实例（这种实例将被传阅，直到信号完成被回收为止） 为副作用使用这些方法---使得事物看起来像主动跟外界(一个没有反应式的世界)交互似的。

- 避免显示状态下进行订阅处理

    - 按照设计准则，无论何时都应该避免显示的订阅处理。请记住我们是怎样用 `takeUntil:` 来自动处理 `FRPCell` 类的订阅的。使用 `takeUntil:` 允许信号值通过，直到它的参数被传递下去或者它自己的值完成。基本上这种情况下，接收者已经完成接收了。
- 内存管理的魔法

    - ARC下，在代码的表面上你摆脱了内存管理。 `ReactiveCocoa` 中也一样。唯一要注意的是，不能在任何signal的block中捕捉self。

以上，就是第五章的全部内容。接下来我们将介绍Model-View-ViewModel这种程序架构，给App添加一个日志系统，并写一些单元测试，出发吧！

> BTY:函数副作用：指当调用函数时，除了返回函数值之外，还对主调用函数产生附加影响。例如修改全局变量或修改参数，一般而言函数副作用会给程序设计带来不必要的麻烦，使程序难以查找错误，并降低程序的可读性。严格的函数式语言要求函数必须无副作用。
>
> 有一种特殊的情况，就是我们这里的函数。它的参数是一种In/Out作用的参数，即函数可能改变参数里面的内容，把一些信息通过输入参数，夹带到外界。这种情况，严格来说，也是副作用，是非纯函数。即我们所讨论的函数反应型编程中的函数式编程属于非纯函数，它是具有副作用的。

MVVM On iOS

有一个禅宗佛教的概念叫做"初心"。禅宗法师铃木俊隆写道："初学者的心中有很多可能性（潜意识的点子），但在专家心里(这种可能性/点子)就相对少很多"。在写作本书的过程中，我经常会回到这个概念里重新审视自己，提醒自己不要对那些看起来很新的或不习惯的事物过早下结论.

本着这种精神，我们回过头来看看你当初接触iOS应用开发的情形：与可能只知道使用Model-View-Controller（MVC）的架构来编写iOS应用的现在的你相比，那时候你一无所知。你的内心随时准备接纳外界无限的可能性（这里指的是任何可以编写iOS应用的方式）。而MVC社区的长老们指导你使用MVC架构来做，因为那就是他们所知道的苹果公司所倡导的方式。

如果你已经用这种方式开发iOS应用程序一段时间，你可能会熟悉MVC背后的另类意义：巨大的视图控制器.(因为MVC:恶搞成Massive View Controller的缩写)。很多时候，我们途方便把业务逻辑和其他代码都放在试图控制器中，即便从架构的角度上来说把它们放在这里不是最佳选择。

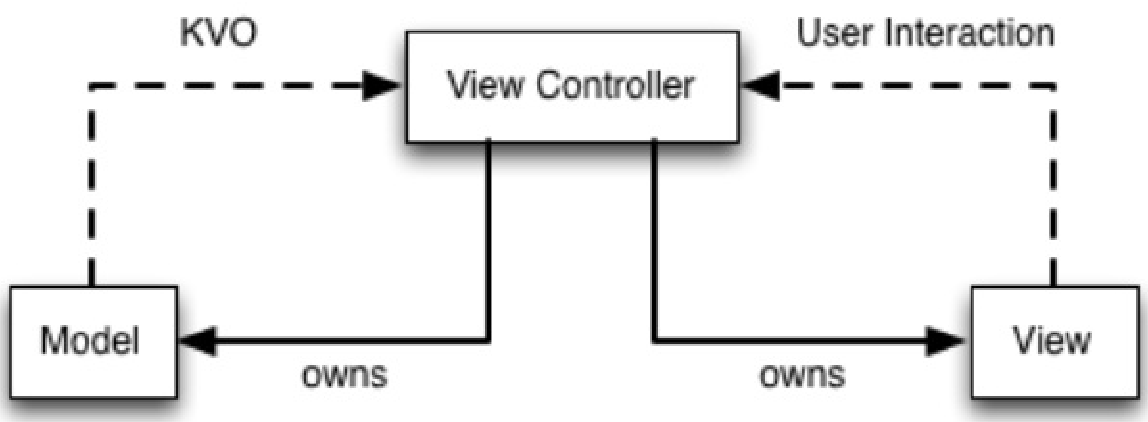
Model View View-Model 也称MVVM，是一种出自微软的替代MVC架构的新架构。我知道，我知道！iOS社区没有任何历史作为微软的铁杆粉丝而存在，但（微软）他们的软件工程小组确实做出了伟大的工作。MVVM不仅仅在.Net平台上使用---我们也可以在iOS平台上使用。就像我们在这一章将要看到的：与ReactiveCocoa结合使用，MVVM令人难以置信地适用于iOS。使用MVVM能够有效地减少ViewController中的业务逻辑，这会大大减少其臃肿的体积，也使得业务逻辑更容易测试。

什么是MVVM

在传统的MVC架构的应用中，你有三种组件：数据模型、视图以及试图控制器。数据模型保持你的数据，而视图用来呈现这些数据。控制器介于这两个组件之间调解所有的交互。

希望于Apple已经很好地测试过它的业务逻辑了。剩下的视图控制器它很少进行单元测试。

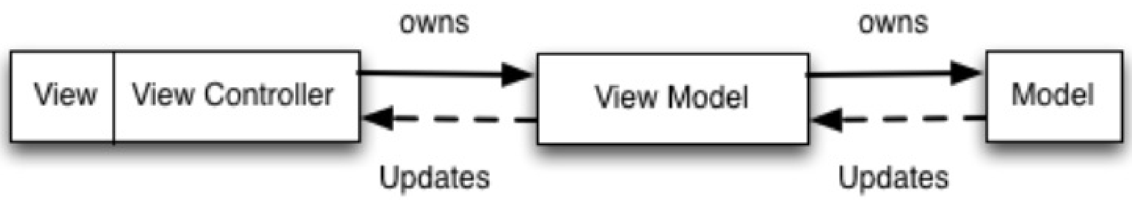
当新的数据到达时，model会通知ViewController（通常是通过键-值观察(KVO)的方式），然后ViewController会更新View。当View接收交互时，ViewController会更新Model。



Typical MVC paradigm

正如你所看到的ViewController隐式地负责很多事情：验证输入、将模型数据映射到面向用户的信息、操作视图层次结构等等。

MVVM将大量的类似上面的业务逻辑从viewController中抽离出来了。



MVVM high level

在MVVM中，我们趋向于将view和view controller作为一个整体（这也解释了为什么不称它为MVVCVM),新的viewModel代替原来的viewController协调view与model之间的交互。

对这种MVVM架构中的"更新"机制，我们没有什么概念。实际上也没有什么关于MVVM的东西迫使你使用特定的机制来更新视图模型或视图。但在本书的范围内，我们将使用ReactiveCocoa来做处理这个。

ReactiveCocoa将会监控数据模型(model)的变化，并将这个变化映射到视图模型(viewModel)的属性上，执行任意必要的业务逻辑。

举一个具体的例子：假设我们的模型包含一个“日期”(用 `dateAdded` 表示)，我们想要监控这个“日期”的变化，来更新我们视图模型(viewModel)的属性 `dateAdded`.模型(model)的属性是一个 `NSDate` 的实例，但视图模型(viewModel)中对应的属性是从它转换过来的 `NSString` 。这种绑定看起来跟下面的代码类似(在viewModel的初始化方法中进行)：

```
RAC(self, dateAdded) = [RACObserve(self.model,dateAdded) map:^(NSDate *date) {
    return [[ViewModel dateFormatter] stringFromDate:date];
}];
```

`dateFormatter` 是ViewModel的一个类方法，它缓存了一个 `NSDateFormatter` 实例以便复用(创建NSDateFormatter代价昂贵)。 接下来，view controller 可以监控viewModel的 `dateAdded` 属性将它跟一个 `label` 进行绑定。

```
RAC(self.label, text) = RACObserve(self.viewModel, dateAdded);
```

现在，我们已经将日期转换为字符串到视图模型的过程抽象出来了，在(viewModel)中我们可以为这个业务逻辑编写单元测试。这个例子看起来简单，但就像我们看到的，它显著地减少了你的视图控制器中的业务逻辑。

重温FunctionalReactivePixels

在我们继续研究使用MVVM来重构我们的 `FunctionalReactivePixels` Demo之前，我们需要做一些准备工作。他们的登陆系统不支持我们使用500px_iOS_SDK的方式。

我们将从AppDelegate的头文件中移除 `apiHelper` 属性，并用下面的代码替换实现文件里执行初始化的那行代码，填上你的消费者Key和Secret.

```
[PXRequest setConsumerKey:consumerKey consumerSecret:consumerSecret];
```

现在，任何调用 `AppDelegate.apiHelper` 来创建500px_API请求的地方，全部必须替换为 `[PXRequest apiHelper]` .

最后，请更新你的CocoaPods文件中500px_iOS_SDK的版本号到 `1.0.5`

MVVM的具体实践

本章的其他部分将把Functional Reactive Pixels Demo的其他代码迁移到MVVM架构中。我们将添加一个新的库到Podfile文件里。Github上创作了ReactiveCocoa的黑客，也同时创建了一个ViewModel的基类:ReactiveViewModel.我们将要使用它的0.1.1版本。更新Podfile之后立即运行 `pod install` 以安装该库。

重构的第一个类是高清图片视图控制器。从这儿开始是因为它的业务逻辑比较少，抽象成viewModel时相对简单。我们循序渐进，慢慢来。

目前，我们的 `FRPFullSizePhotoViewController` 包含一个图片数组和当前图片(在数组中)的下标值。我们将把他们抽象到我们的视图模型中来。

从头文件中移除自定义初始化，追加 `FRPFullSizePhotoViewModel` 的预申明。然后在这个新类中追加一个属性。

```
@property (nonatomic ,strong ) FRPFullSizePhotoViewModel *viewModel;
```

在实现文件里，#import这个新的视图模型(别担心，我们很快就会创建它)，

```
#import "FRPFullSizePhotoViewModel.h"
```

然后，移除 `photoModelArray` 私有属性的申明。重写我们的初始化方法以移除对 `photoModelArray` 实例的引用。代码看起来应该像下面这样:

```
- (instancetype)init {
    self = [super init];
    if(!self) return nil;

    //ViewControllers
    self.pageViewController = [UIPageViewController alloc]
                    initWithTransitionStyle:UIPageViewControllerTransitionStyleScroll
                    navigationOrientation:UIPageViewControllerNavigationOrientationHorizo
                       options:@{ UIPageViewControllerOptionInterPageSpacingKey : @30 };

    self.pageViewController.dataSource = self;
    self.pageViewController.delegate      = self;
    [self addChildViewController:self.pageViewController];

    return self;
}
```

在你的 `ViewDidLoad:` 中添加如下代码：

```
//Configure child view controllers
```

```
[self.pageViewController \
        setViewControllers: @[ [self photoViewControllerForIndex:self.viewModel.initialPh
        direction:UIPageViewControllerNavigationDirectionForward
        animated:NO
        completion:nil ];

//Configure self
self.title = [self.viewModel.initialPhotoModel photoName];
```

我们将要写的这个我们提到的方法，对于veiwModel中发生的事情，给你一种XX感。最后，进到`photoViewControllerForIndex`方法中，它应用了已经解除分配的`photoModelArray`，用下面的实现替代它。

```
- (FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index {
    if (index >= 0 && index < self.viewModel.photoArray.coung ) {
        FRPPhotoModel *photoModel = self.viewModel.model[index];

        FRPPhotoViewController *photoViewController = \
            [[FRPPhotoViewController alloc] initWithPhotoModel:photoModel index:index];

        return photoViewController;
    }

    // Index was out of bounds, return nil
    return nil;
}
```

好了！现在轮到我们的视图模型本身了。创建一个新的`RVMViewModel`的子类，并将其命名为`FRPFullSizedPhotoViewModel`.基于它将要封装的信息，以及我们在视图控制器中的需求，我们知道，我们的头文件看起来应该是下面这样：

```
@class FRPPhotoModel;

@interface FRPFullSizePhotoViewModel : RVMViewModel

- (instancetype)initWithPhotoArray:(NSArray *)photoArray initialPhotoIndex:(NSInteger)ini
- (FRPPhotoModel *)photoModelAtIndex:(NSInteger)index;

@property (nonatomic , readonly, strong) NSArray *model;
@property (nonatomic, readonly) NSInteger initialPhotoIndex;
@property (nonatomic, readonly) NSString *initialPhotoName;

@end
```

`model`属性在`RVMViewModel`中被定义为`id`类型，我们把它重定义为`NSArray`. 我们也勾住了(即使用全局变量记录)我们最初照片的索引(下标)并且给我们最初的照片名属性定义了只读属性。这种微不足道的逻辑我们可以放到我们的视图控制器中，但很快我们就会看到更为复杂的情况。

我们来完成实现文件里的东西。第一件事就是：我们需要#import `FRPPhotoModel`类的头文件。然后，我

们将打开私有属性的读写访问权限。

```
//Model
#import "FRPPhotoModel.h"

@interface FRPFullSizePhotoViewModel ()
//private access
@property (nonatomic, assign) NSInteger initialPhotoIndex;

@end
```

好！下一步处理我们的初始化方法

```
- (instancetype)initWithPhotoArray:(NSArray *)photoArray initialPhotoIndex:(NSInteger)ini
    self = [super initWithModel:photoArray];
    if(!self) return nil;

    self.initialPhotoIndex = initialPhotoIndex;

    return self;
}
```

初始化方法中，先调用超类的`initWithModel:`实现，然后设置自己的`initialPhotoIndex`属性。剩下的两个只读属性的获取逻辑微不足道。

```
- (NSString *)initialPhotoName {
    return [[self photoModelAtIndex:self.initialPhotoIndex] photoName];
}

- (FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
    if(index < 0 || index > self.model.count - 1) {
        //Index was out of bounds, return nil
        return nil;
    }
    else {
        return self.model[ index ];
    }
}
```

这样做的另一个优点是：业务逻辑不需要重复书写，而且也使得业务逻辑非常好进行单元测试。

最后，我们需要在高清视图控制器中设置该视图模型，否则屏幕上将不会显示任何东西。导航到我们的画廊视图控制器（那个我们实例化并推出高清视图控制器的地方）。用下面的代码来替换这个业务逻辑：

```
[[self rac_signalForSelector:@selector(collectionView:didSelectItemAtIndexPath:)
    fromProtocol:@protocol(UIcollectionViewDelegate)] subscribeNext:^(RACTuple *arguments
        @strongify(self);

        NSIndexPath *indexPath = arguments.second;
        FRPFullSizePhotoViewModel *viewModel = [[FRPPhotoViewModel alloc]
```

```
            initWithPhotoArray:self.viewModel.model initialPhotoIndex:indexPath.item];

        FRPFullSizePhotoViewController *viewController = [[FRPFullSizePhotoViewController

        viewController.viewModel = viewModel;
        viewController.delegate = (id)self;

        [self.navigationController pushViewController:viewController animated:YES];
    }];
```

在下一节开始之前，我们没有计划为视图模型撰写单元测试。下一节我们看到在视图模型上如何运行测试驱动开发的概念。现在我们来完成 `FRPGalleryViewModel` 吧，很基础。我们想要从视图控制器中抽象出来的逻辑是通过API加载 `model` 的数据内容。我们来看一下应该怎么做：

```
@interface FRPGalleryViewModel : RVMViewModel

@property (nonatomic, readonly, strong) NSArray *model;

@end
```

基本的接口：将 `model` 申明为数组 `NSArray` .接下来，我们简单实现它：

```
//Utilities

#import "FRPPhotoImporter.h"

@interface FRPGalleryViewModel ()

@end

@implementation FRPGalleryViewModel

- (instancetype)init {
    self = [super init];
    if(!self) return nil;

    RAC(self, model) = [[[FRPPhotoImporter importPhotos] logError] catchTo:[RACSignal emp

    return self;
}

@end
```

有争议的是，我们应该把从API加载数据的(RAC绑定的)逻辑放在初始化方法中，还是放在视图模型被激活的地方。接下来我们会讨论更多的关于激活的内容，但我想要展示给你们看这个视图模型到底能做到多简单。将直接在画廊视图控制器中加载数据内容的逻辑迁移到画廊的视图模型中是非常简单的：在视图控制器的初始化中初始化视图模型===》任何引用试图控制 `self.model` 属性的地方使用 `self.viewModel.model` 来代替即可。

我们可以进一步深挖视图模型的构造，甚至可以通过一系列的访问器把`model`的访问逻辑抽象出来，但在这个例子里就有点过多'抽象'了。更重要的是你可以根据你的喜好将更多的或者更少的业务逻辑抽象到视图模型中。我发现，就我个人而言，这个架构使用的越多，业务逻辑抽象出来的越多，就意味着更轻量级的视图控制器以及高内聚和可测试的代码。

把注意力移到单元测试之前，我们来做多一次用视图模型来抽象业务逻辑的实践。

我们的最后一个例子是`FRPPhotoViewController`上的`FRPPhotoViewModel`:创建一个`RVMViewModel`的视图模型子类并放置在视图控制器中(很快我们会回到视图模型中)。

视图控制器的新的初始化方法如下：

```
- (instancetype)initWithViewModel:(FRPPhotoViewModel *)viewModel index:(NSInteger)photoIn

    self = [self init];//NS_DESIGNATED_INITIALIZER
    if(!self) return nil;

    self.viewModel = viewModel;
    self.photoIndex = photoIndex;

    return self;
}
```

确定导入必要的头文件并为视图模型申明私有属性。现在我们需要使用新的初始化方法初始化视图控制器。看一看视图控制器到页面视图控制器的方法`photoViewControllerForIndex:`.

```
- (FRPPhotoViewController *)photoViewControllerForIndex:(NSInteger)index {
    FRPPhotoModel *photoModel = [self.viewModel photoModelAtIndex:index];
    if(photoModel) {
        FRPPhotoViewModel *photoViewModel = [[FRPPhotoViewModel alloc] initWithModel:phot
        FRPPhotoViewController *photoViewController = [[FRPPhotoViewController alloc] \
                              initWithViewModel:photoViewModel
                                          index:index];

        return photoViewController;
    }

    return nil;
}
```

新的初始化过程中我们创建了一个视图模型。

在我们的`viewDidLoad:`方法里，我们将使用这个新的视图模型为我们的图片视图提供数据，并且为用户显示图片的下载进度。这里有个貌似冲突的地方：图片的下载是视图的模型的业务逻辑之一，但视图什么时候显示开始加载数据(这个业务逻辑)视图模型中没有体现---记住一个好的视图模型不应该引用视图本身。那么我们如何来混合地使用这两个业务逻辑？

答案是我们借助视图模型的`active`状态来对付（上面的情况）。`RVMViewModel`提供了一个布尔属性`active`，当试图控制器变得"活跃"时(不管在语义的上下文里这是啥意思)，在这里，我们可以

在`viewWillAppear:`和`viewDidDisappear:`这些方法来设置这个属性。

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    self.viewModel.active = YES;
}

- (void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];

    self.viewModel.active = NO;
}
```

相当简单吧，我们来看一下我们新的`viewDidLoad`方法：

```
- (void)viewDidLoad {
    [super viewDidLoad];

    //Configure self's view
    self.view.backgroundColor = [UIColor blackColor];

    //Configure subViews
    UIImageView *imageView = [[UIImageView alloc] initWithFrame:self.view.bounds];
    RAC(imageView, image) = RACObserve(self.viewModel,photoImage);
    imageView.contentModel = UIViewContentModelScaleAspectFit;
    [self.view addSubView:imageView];
    self.imageView = imageView;

    [RACObserve(self.viewModel, loading) subscribeNext:^(NSNumber *loading) {
        if(loading.boolValue) {
            [SVProgressHUD show];
        }
        else {
            [SVProgressHUD dismiss];
        }
    }];
}
```

该图片视图的图片属性的绑定是标准的ReactiveCocoa方式,有趣的是下面(我们要提到的)我们使用`loading`的时刻。当加载信号发送`YES`的时候我们展示进度HUD，发送`NO`的时候，让进度HUD消失。我们将看到该`loading`信号本身如何依赖于`didBecomeActiveSignal`。现在只是视图模型通过网络请求获取图像数据的序幕。

接口的申明如下：

```
@class FRPPhotoModel;

@interface FRPPhotoViewModel : RVMViewModel

@property (nonatomic, readonly) FRPPhotoModel *model;
@property (nonatomic, readonly) UIImage *photoImage;
@property (nonatomic, readonly, getter = isLoading) BOOL loading;
```

```
- (NSString *)photoName;

@end
```

该 `model` 和 `photoImage` 属性的用法已经解释过了。 `photoName` 事实上作为属性在代码库的其他地方被用来设置一些东西，类似于分页视图控制器的标题这样。你可以下载Github的代码库了解详情。我们来看一下实现：

```
#import "FRPPhotoViewModel.h"

//Utilities
#import "FRPPhotoImporter.h"
#import "FRPPhotoModel.h"

@interface FRPPhotoViewModel ()

@property (nonatomic, strong) UIImage *photoImage;
@property (nonatomic, assign, getter = isLoading) BOOL loading;

@end

@implementation FRPPhotoViewModel

- (instancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if(!self) return nil;

    @weakify(self);
    [self.didBeComeActiveSignal subscribeNext:^(id x) {
        @strongify(self);
        self.loading = YES;
        [[FRPPhotoImporter fetchPhotoDetails:self.model] subscribeError:^(NSError *error)
            NSLog(@"Could not fetch photo details: %@",error);
        } completed:^{
            self.loading = NO;
            NSLog(@"Fetched photoDetails.");
        }];
    }];

    RAC(self, photoImage) = [RACObserve(self.model, fullsizedData) map:^id (id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
}

- (NSString *)photoName {
    return self.model.photoName;
}

@end
```

该 `didBecomeActive` 信号订阅带有"函数副作用"的加载照片详情包括它的高清图片的数据。然

后 `photoImage` 属性与模型的映射结果绑定。

使用 `didBecomeActiveSignal` 这种方法来启动一些像网络操作这样昂贵的任务，远远优于我们早前在初始化方法中启动他们的方法。

这就是在本书中我们将要涉及的全部内容，更多详情请参考functional reactive pixels，这个代码库包含了更多的在图片详情视图控制器和登陆视图控制器中使用视图模型的例子。这些Demo将向你展示如何有效地使用 `ReactiveCocoa` 执行网络操作和使用 `RACCommands` 响应用户界面交互。

测试ViewModels

本书的最后一节，我们谈谈测试，尤其是单元测试。在iOS的开发社区里，这是一个有争议的话题，这也是为什么我要把它放在最后的原因。理想的情况下。你应该在编写视图模型的同时为它编写单元测试。然而学习如何使用这种新的模式来编码已经很困难，尝试去测试这些你没有吃透的东西，多你来说压力太大，所以我把它放在了最后（学到这里我相信你已经理解了这种编码方式）。

当然我也注意到，并不是每个人都以相同的方式来测试，或者能够测试到相同的程度。我有.Net编程背景，在.net中使用mocks来测试系统的实现细节是最平常不过的了。其他平台背景的开发者较少使用mocks来做，甚至从来没有这样的经验。本节我只将我的单元测试方法分享给大家，如果你觉得合适就采用。

确保你的`Podfile`文件包含下面这些库：

```
target "FRPTests" do

pod 'ReactiveCocoa', '2.1.4'
pod 'ReactiveViewModel', '0.1.1'
pod 'libextobjc', '0.3'
pod '500px-iOS-api', '1.0.5'
pod 'Specta', '~> 0.2.1'
pod 'Expecta', '~> 0.2'
pod 'OCMock', '~> 2.2.2'

end
```

然后运行`pod install`.

首先我们来看看`FRPFullSizePhotoViewModel`，因为它最具Objective-C风范(没有太多ReactiveCocoa).

```
@interface FRPFullSizePhotoViewModel ()
//Private access
@property (nonatomic, assign) NSInteger initialPhotoIndex;

@end

@implementation FRPFullSizePhotoViewModel

- (instancetype)initWithPhotoArray:(NSArray *)photoArray initialPhotoIndex:(NSInteger)ini
    self = [self initWithModel:photoArray];
    if(!self) return nil;

    self.initialPhotoIndex = initialPhotoIndex;

    return self;
}

- (NSString *)initialPhotoName {
    return [self.model[self.initialPhotoIndex] photoName];
}

- (FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
```

```
    if(index < 0 || index > self.model.count - 1) {
        //Index was out of bounds, return nil
        return nil;
    }
    else {
        return self.model[index];
    }
}

@end
```

好了，我们先来测试这个初始化方法，然后在转移到其他两个方法上。

我们想印证初始化我们的视图模型时，它的两个属性 `model` 和 `initialPhotoIndex` 被正确地赋值了。

```
#import
#define EXP_SHORTHAND
#import
#import
#import "FRPPhotoModel.h"

#import "FRPFullSizePhotoViewModel.h"

SpecBegin(FRPFullSizePhotoViewModel)

describe(@"FRPFullSizePhotoModel", ^{
    it (@"Should assign correct attributes when initialized", ^{
        NSArray *model = @[];
        NSInteger initialPhotoIndex = 1337;

        FRPFullSizePhotoViewModel *viewModel =\
         [[FRPFullSizePhotoViewModel alloc] initWithPhotoArray:model
                                               initialPhotoIndex: initialPhotoIndex

        expect(model).to.equal(viewModel.model);
        expect(initialPhotoIndex).to.equal(viewModel.initialPhotoIndex);

    });
});

SpecEnd
```

在该代码段顶部，我们导入了一些头文件，包括一个奇怪的预定义 `EXP_SHORTHAND` ,我们把他放在那里以便于可以使用类似 `expect()` 这样的shorthand matchers（速记匹配）的语法。然后我们引入我们的私有接口 `SpecBegin(...)/SpecEnd` 来为我们正在测试的视图模型屏蔽编译警告，最后的部分就是我们的单元测试本身。 `Specta` 的测试规范相当简单，你可以阅读更多的关于这方面的信息，但本书不会深入讲解它的一些细节。总之你的测试始于 `SpecBegin` 并终止于 `SpecEnd` ，测试例程用类似于 `@"应该。。。",^{ 预测正常的情况应该如何 }` 写在中间。

好了，停止模拟器中正在运行的应用，按下 `cmd+U` 快捷键，你就可以运行这段单元测试了。如果一切正常，你就能通过测试。

接下来我们来看看 `photoModelAtIndex:` 方法

```
- (FRPPhotoModel *)photoModelAtIndex:(NSInteger)index {
    if(index < 0 || index > self.model.count - 1 ) {
        // Index was out of bounds ,return nil
        return nil;
    }
    else {
        return self.model[ index ];
    }
}
```

这里面没有太多的业务逻辑，但是我们看到其他地方都要使用它，所以我们的测试应该是健壮的。

```
it(@"Should return nil for an out-of-bounds photo index", ^{
    NSArray *model = @[[NSobject new]];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel = \
        [[FRPFullSizePhotoViewModel alloc] initWithPhotoArray:model initialPhotoIndex:ini

    id subzeroModel = [viewModel photoModelAtIndex:-1];
    expect(subzeroModel).to.beNil();

    id aboveBoundsModel = [viewModel photoModelAtIndex:model.count];
    expect(aboveBoundsModel).to.beNil();
});

it(@"Should return the correct model for photoModelAtIndex:",^{
    id photoModel = [NSObject new];
    NSArray *model = @[photoModel];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel = \
        [[FRPFullSizePhotoViewModel alloc] initWithPhotoArray:model initialPhotoIndex:ini

    id returnModel = [viewModel photoModelAtIndex:0];
    expect(returnModel).to.equal(photoModel);

});
```

太棒了！我们这个新的测试保证了我们的代码具有完全的代码覆盖率。它检测了 `photoModelAtIndex:` 参数的三种可能的情况：少于0、在作用范围内以及越界。

最后，我们来看下 `initialPhotoName` 方法：

```
- (NSString *)initialPhotoName {
    return [self.model[self.initialPhotoIndex] photoName];
}
```

方法看起来很简单，但实际上这里面包含了更深层级的东西。恰当地重构一些代码并为它写一点不一样的

更小的测试代码，来严格地测试这个方法。

```
- (NSString *)initialPhotoName {
    FRPPhotoModel *photoModel = [self initialPhotoModel];
    return [photoModel photoName];
}

- (FRPPhotoModel *)initialPhotoModel {
    return [self photoModelAtIndex:self.initialPhotoIndex];
}
```

这更清晰简单了，一个方法确切地只做一件事情，就像一棵树的树皮，层层叠叠相互依存。只要我们一路下来所有的代码都测试，那么最后我们就可以很确切地保证代码的健壮性。

`initialPhotoModel`是一个私有方法，所以测试它我们需要在测试文件中申明它。

```
@interface FRPFullSizePhotoViewModel ()

- (FRPPhotoModel *)initialPhotoModel;

@end
```

你看到的所有我们的测试代码都非常简单。

```
it (@"Should return the correct initial photo model", ^{
    NSArray *model = @[[NSObject new]];
    NSInteger initialPhotoIndex = 0;

    FRPFullSizePhotoViewModel *viewModel = \
        [[FRPFullSizePhotoViewModel alloc] initWithPhotoArray:model initialPhotoIndex:ini

    id mockViewModel = [OCMockObject partialMockForObject:viewModel];
    [[[mockViewModel expect] andReturn:model[0]] photoModelAtIndex:initialPhotoIndex];

    id returnedObject = [mockViewModel initialPhotoModel];

    expect(returnedObject).to.equal(model[0]);

    [mockViewModel verify];
});
```

这个测试是用来确认当`initialPhotoModel`被调用时，接下来它应该调用`photoModelAtIndex:`方法并将`initialPhotoIndex`作为参数传入。这个测试是否简单取决于我们测试`photoModelAtIndex:`是否充分。

接下来，就让我们一起来看看`FRPGalleryViewModel`,这看似非常简单：

```
- (instancetype)init {
    self = [super init];
```

```
    if(!self) return nil;

    RAC(self, model) = [[[FRPPhotoImporter importPhotos] logError] catchTo:[RACSignal emp

    return self;
}
```

然而，它可测性不高，需要重构。

我们简单地重构下视图模型。新的实现如下：

```
@implementation FRPGalleryViewModel

- (instancetype)init {
    self = [super init];
    if(!self) return nil;

    RAC(self, model) = [self importPhotosSignal];

    return self;
}

- (RACSignal *)importPhotosSignal {
    return [[[FRPPhotoImporter importPhotos] logError] catchTo:[RACSignal empty]];
}

@end
```

我们把 `importPhotos` 的调用抽出来，以方便测试这个方法是否被调用。我们不会测试 `FRPPhotoImporter` ，关于它的测试(即单例测试)已经超出了本书的范畴。

这部分的测试代码如下：

```
#import "Specta.h"
#import

#import "FRPGalleryViewModel.h"

@interface FRPGalleryViewModel ()

- (RACSignal *)importPhotosSignal;

@end

SpecBegin(FRPGalleryViewModel)

describe(@"FRPGalleryViewModel",^{
    it(@"should be initialized and call importPhotos", ^{
        id mockObject = [OCMockObject mockForClass:[FRPGalleryViewModel class]];
        [[[mockObject expect] andReturn:[RACSignal empty]] importPhotosSignal];

        mockObject = [mockObject init];
```

```
        [mockObject verify];
        [mockObject stopMocking];
    });
});
```

为了测试一个方法，测试代码也太多了吧！ 我知道，我知道~ 这是OCMock没落的原因之一，它竟然需要这么多的模板。但你不能责怪它，因为它要工作在令它不寒而栗的Objective-C平台上！

我们创建了一个`FRPGalleryViewModel`的mock版本，告诉它期望`importPhotoSignal`被调用。然后才进行对象的初始化。这里使用了一点点技巧，因为我们在mockObject上调用了init方法，但它(init)实际上是一个NSProxy的子类。然后，对OCMock来讲，它足够聪明，它了解这一切，有能力做出正确的选择。只是看起来有点诡异罢了。我们使用`[mockObject init]`给`mockObject`赋值，也是为了屏蔽编译警告。最后我们验证了所有预期可能被调用的方法。

这个例子中表现出来的测试很困难的情况也说明了另一个问题，你应该避免视图模型的初始化方法产生"副作用"(参见前面章节提到的“函数的副作用”)，应该使用`didBecomeActiveSignal`来代理。

下面我们来测试`FRPPhotoViewModel`.再次突出引起函数副作用和使用`didBecomeActiveSignal`的区别。

快速浏览下实现：

```
@implementation FRPPhotoViewModel

- (intancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if(!self) return nil;

    @weakify(self);
    [self.didBecomeActiveSignal subscribeNext:^ (id x) {
        @strongify(self);
        self.loading = YES;
        [[FRPPhotoImporter fetchPhotoDetails:self.model]
            subscribeError: ^ (NSError *error) {
                NSLog(@"Could not fetch photo details: %@",error);
            }
            completed: ^ {
                self.loading = NO;
                NSLog(@"Fetched photo details");
            }];
    }];

    RAC(self, photoImage) = [RACObserve(self.model, fullsizedData) map:^id (id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
}

- (NSString *)photoName {
    return self.model.photoName;
}

@end
```

首先我们来测试 `photoName` 方法：

```
#import
#define EXP_SHORTHAND
#import
#import

#import "FRPPhotoViewModel.h"
#import "FRPPhotoModel.h"

SpecBegin(FRPPhotoViewModel)

describe (@"FRPPhotoViewModel", ^{
    it(@"should return the photo's name property when photoName is invoked", ^{
        NSString *name = @"Ash";

        id mockPhotoModel = [OCMockObject mockForClass:[FRPPhotoModel class]];
        [[[mockPhotoModel stub] andReturn:name] photoName];

        FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:nil];
        id mockViewModel = [OCMockObject partialMockForObject:viewModel];
        [[[mockViewModel stub] andReturn:mockPhotoModel] model];

        id returnName = [mockViewModel photoName];

        expect(returnedName).to.equal(name);
        [mockPhotoModel stopMocking];
    });
});
```

我们为mock的视图模型的model属性添加了一个mockPhotoModel，它会mocks所有的途径。

现在来看这个复杂的初始化方法，这东西看起来真巨大！近20行纯粹的未经测试的代码。哎呀！让我们来一点点简化这个事情，并逐步加上我们的测试代码。

```
- (instancetype)initWithModel:(FRPPhotoModel *)photoModel {
    self = [super initWithModel:photoModel];
    if(!self) return nil;

    @weakify(self);
    [self.didBecomeActiveSignal subscribeNext:^(id x) {
        @strongify(self);
        [self downloadPhotoModelDetails];
    }];

    RAC(self, photoImage) = [RACObserve(self.model, fullsizedData) map:^id (id value) {
        return [UIImage imageWithData:value];
    }];

    return self;
}

- (void)downloadPhotoModelDetails {
```

```
    self.loading = YES;
    [[FRPPhotoImporter fetchPhotoDetails:self.model] subscribeError:^(NSError *error) {
        NSLog(@"Could not fetch photo details : %@",error);
    } completed:^ {
        self.loading = NO;
        NSLog(@"Fetched photo details.");
    }];
}
```

我们选择了不直接测试`fetchPhotoDetails:`,所以我们把它置于一个实例方法中，以便更容易对它进行测试。这个方法(即`fetchPhotoDetails:`)实现的细节在这里对我们不重要。

现在开始写关于它的测试代码吧：

```
it(@"should download photo model details when it becomes active", ^{
    FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:nil];

    id mockViewModel = [OCMockObject partialMockForObject:viewModel];
    [[mockViewModel expect] downloadPhotoModelDetails];

    [mockViewModel setActive:YES];
    [mockViewModel verify];
});
```

注意看初始化方法中不产生(函数)副作用而是把这种副作用放在订阅`didBecomeActiveSignal`的Block块中时，测试视图模型的代码是多么简单！

现在我们需要测试剩下的那些视图模型，他们全部非常简单。我们使用更少的mock，因为很多的业务逻辑仅仅是视图模型的model值到他自己的属性的映射。

```
it (@"should return the photo's name property when photoName is invoked", ^{
    NSString *name = @"Ash";

    id mockPhotoModel = [OCMockObject mockForClass:[FRPPhotoModel class]];
    [[[mockPhotoModel stub] andReturn:name] photoName];

    FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:nil];
    id mockViewModel = [OCMockObject partialMockForObject:viewModel];
    [[[mockViewModel stub] andReturn:mockPhotoModel] model];

    id returnedName = [mockViewModel photoName];

    expect(returnedName).to.equal(name);

    [mockPhotoModel stopMocking];
});

it (@"should correctly map image data to UIImage", ^{
    UIImage *image = [[UIImage alloc] init];
    NSData *imageData = [NSData data];

    id mockImage = [OCMockObject mockForClass:[UIImage class]];
    [[[mockImage stub] andReturn:image] imageWithData:imageData];
```

```
    FRPPhotoModel *photoModel = [[FRPPhotoModel alloc] init];

    photoModel.fullsizedData = imageData;

    __unused FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:phot

    [mockImage verify];
    [mockImage stopMocking];

});

it(@"should return the correct photo name", ^{
    NSString *name = @"Ash";

    FRPPhotoModel *photoModel = [[FRPPhotoModel alloc] init];
    photoModel.photoName = name;

    FRPPhotoViewModel *viewModel = [[FRPPhotoViewModel alloc] initWithModel:photoModel];

    NSString *returnedName = [viewModel photoName];

    expect(name).to.equal(returnedName);
});
```

这就是为视图模型撰写单元测试的全部内容了。

在理想的情况下，单元测试能帮助改进你的代码质量。小巧而高内聚的方法比随意的满是副作用的方法更招人待见。它简单而完美地诠释了函数响应型编程的精髓。

测试MVVM的好处是：我们不用触及UIKit。请记住，写得好的MVVM视图模型的特点是：该视图模型不会与用户交互的接口类有任何交互。

终稿

MVVM是一个非常有趣的架构。在这方面我思考的越多，它对我的意义便越大。诚然，本章中的视图模型所呈现的业务逻辑都很轻量。我已经把它们上传到Github仓库里了，但是本章作为一个MVVM的示例为初学者的开始提供了参考。

我想提供一个具体的例子来说明它比MVC更有竞争力，更具意义。

最近我创建的一个App中，我们有一堆数据，支持下拉刷新,每一个元素点击之后会推出详情页面,视图控制器有很多业务逻辑,非常标准的东西。然而，这一堆数据彼此之间来路是不一样的，有的是主API入口的数据结果，有的是它们的搜索结果，有的是App在编译时就决定的静态元素。

使用MVC的话，我想到了两种方法来解决：

1. 在臃肿的视图控制器中创建一个类处理显示,并管理所有的数据内容
2. 毫无疑问，另一种方法就是子类化一个视图控制器的抽象基类来包含所有内容的通用逻辑。

这是过去我所采用的方式，但这方式很难重构，比方说：有些所有类型的通用内容变得只对部分类型有效时。这同样也能被称为是一种黑客攻击，因为Objective-C不支持抽象类。

我采用的方法是使用符合该视图控制器所依赖的协议的不同的视图模型。通过将定制的业务逻辑放置于视图模型中，我避免了视图控制器的臃肿化，视图控制器仅需要根据视图模型的协议来知道如何显示即可。MVVM是子类化视图控制器的一个很好的选择。

另外，如果你有多平台需求(比如说：iOS & OSX)，他们可以共用一套视图模型，因为他们不牵扯到视图本身的逻辑。你甚至可以走得更远，用另外的语言来生成视图模型，然后生成指定的语言的视图模型对象比如：Objective-C、C#、Ruby、Java或者其他你需要的任何语言。疯狂吧这玩意~

最后，我们并没有真正地涉及到 `RACCommand` 的使用。我将利用Justin Spahr-Summers的说法在MVVM的范畴来解释它。

1. 控制(事件)与它交互
2. 一个属于视图模型的命令被执行
3. 视图模型的逻辑被运行(运行的是命令初始化时的signalBlock)
4. 视图模型通过ReactiveCocoa来间接通知视图。在我们的例程中，视图会被更新。

再一次强调Github仓库包含了我们在本书中没有能够涉及的，关于 `RACCommand` 的，使用的详细信息。去看一看吧！

MVVM效果很好，与ReactiveCocoa结合起来使用更好。你没有必要一下子就被它“招安”了。你可以从小处着手，先在一个视图控制器中使用，看看你到底能有多喜欢它。在你的下一个项目中尝试使用它把，你会看到它如何彻底简化你的视图控制器的复杂度。